TITLE Title Page

Object-Oriented Application Development with VisualAge in a Client/Server Environment

Document Number GG24-4227-00

June 1994

International Technical Support Organization
San Jose Center Center

NOTICES Notices

```
+--- Take Note! -----------------------------------------------------+
¦                                                                    ¦
¦ Before using this information and the product it supports, be sure ¦
¦ to read the general information under "Special Notices" in         ¦
¦ topic FRONT_1.                                                     ¦
¦                                                                    ¦
+--------------------------------------------------------------------+
```

EDITION Edition Notice

First Edition (June 1994)

This edition applies to Version 1.0 of VisualAge Team 1.0 87G7049 for use with the OS/2 operating system.

Order publications through your IBM representative or the IBM branch office serving your locality. Publications are not stocked at the address given below.

An ITSO Technical Bulletin Evaluation Form for reader's feedback appears facing Chapter 1. If the form has been removed, comments may be addressed to:

IBM Corporation, International Technical Support Organization
Dept. 471, Building 070B
5600 Cottle Road
San Jose, California 95193-0001

ABSTRACT Abstract

This document provides guidelines for the development of object-oriented applications using VisualAge, a robust visual programming tool developed at the IBM Cary Laboratory.

The guidelines are presented in the context of the development of an object-oriented banking application using VMT, a methodology designed for object-oriented visual programming environments, such as VisualAge. The methodology includes use case analysis, prototyping, and GUI building. IBM VisualAge customers with large MIS departments have used VMT successfully.

This publication is written for software development managers, software designers and application developers who plan to develop Client/Server computing applications using VisualAge. Some knowledge of object-oriented modeling and the VisualAge product is assumed.

AD

Subtopics

ABSTRACT.1 Acknowledgments

This document is the result of a residency project run at the International Technical Support Organization - San Jose Center, from October through December 1993. This project was designed and managed by:

Daniel S. Tkach	International Technical Support Organization - San Jose Center.

The authors of this document are:

Walter Fang	IBM Canada - Toronto
Andrew C. So	IBM China - Hong Kong
Alessandro Mottadelli	IBM Italy - Milan
Daniel Tkach	IBM ITSO - San Jose Center
Thomas K. Donahue	IBM US - San Diego

Thanks to the following people for the invaluable support and guidance they provided in the production of this document:

Martine Jhabvala	IBM Cary Laboratory
Martin Nally	IBM Cary Laboratory
Johan Stäbler	International Technical Support Organization - San Jose Center

Thanks are also due to:

Kirk Davis	IBM Santa Teresa Laboratory
Alistair Cockburn	IBM OO Consulting

and to the many other collaborators and reviewers who contributed to improve this document.

CONTENTS Table of Contents

FIGURES Figures

TABLES Tables

FRONT_1 Special Notices

This publication is intended to help software development managers, software designers, and application developers design and develop Client/Server computing applications using VisualAge. The information in this publication is not intended as the specification of any programming interfaces that are provided by VisualAge. See the PUBLICATIONS section of the IBM Programming Announcement for VisualAge for more information about what publications are considered to be product documentation.

References in this publication to IBM products, programs or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM product, program, or service is not intended to state or imply that only IBM's product, program, or service may be used. Any functionally equivalent program that does not infringe any of IBM's intellectual property rights may be used instead of the IBM product, program or service.

Information in this book was developed in conjunction with use of the equipment specified, and is limited in application to those specific hardware and software products and levels.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to the IBM Director of Commercial Relations, IBM Corporation, Purchase, NY 10577.

The information contained in this document has not been submitted to any formal IBM test and is distributed AS IS. The information about non-IBM (VENDOR) products in this manual has been supplied by the vendor and IBM assumes no responsibility for its accuracy or completeness. The use of this information or the implementation of any of these techniques is a customer responsibility and depends on the customer's ability to evaluate and integrate them into the customer's operational environment. While each item may have been reviewed by IBM for accuracy in a specific situation, there is no guarantee that the same or similar results will be obtained elsewhere. Customers attempting to adapt these techniques to their own environments do so at their own risk.

Any performance data contained in this document was determined in a controlled environment, and therefore, the results that may be obtained in other operating environments may vary significantly. Users of this document should verify the applicable data for their specific environment.

The following document contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples contain the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

Reference to PTF numbers that have not been released through the normal distribution process does not imply general availability. The purpose of including these reference numbers is to alert IBM customers to specific information relative to the implementation of the PTF when it becomes available to each customer according to the normal IBM PTF distribution process.

The following terms, denoted by an asterisk (*) in this publication, are trademarks of the IBM Corporation in the United States and/or other countries:

IBM	AD/Cycle
APPC	AS/400
BookMaster	CICS ECI
CICS OS/2	CUA
DB2/2	DRDA
ECI	MVS/ESA
OS/2	RISC System/6000
SAA	System/390
VisualAge	VM/ESA

The following terms, denoted by a double asterisk (**) in this publication, are trademarks of other companies:

CommonView	Glockenspiel
ENVY	Object Technology International, Inc
Excelerator II	Intersolv
GemStone	Servio Corporation
MacApp	Apple Computer, Inc.
Objectivity/DB	Objectivity, Inc.
ObjectStore	Object Design International
ObjectWindows	Borland
Ontos	Ontos Corporation

OMTool	GE Advanced Concept Center
OpenODB	Hewlett-Packard Corporation
Visual Basic	Microsoft Corporation
Visual C++	Microsoft Corporation
VXRexx**	WATCOM
Powerbuilder	Powersoft
VisualWorks	ParcPlace
Enfin	Easel
PARTS Workbench	Digitalk
Windows	Microsoft Corporation

PREFACE Preface

This document describes the process of designing and building an object-oriented application with VisualAge, an integrated and robust development environment. It contains:

- An object-oriented analysis and design methodology for building applications with VisualAge
- Sample application components with supporting documentation to illustrate the design and coding
- Recommendations for building object-oriented applications with VisualAge.

This document is intended for software engineers, application designers, application programmers, and software development managers.

Subtopics

PREFACE.1 Document Organization

The document is organized as follows:

Part 1: Introduction

- Chapter 1, "The Visual Age of Application Development"

  This chapter provides an introduction to visual programming and VisualAge.

- Chapter 2, "Client/Server Computing and Object Technology"

  This chapter provides an overview of how object technology relates to Client/Server computing.

- Chapter 3, "Planning an Object-Oriented Development Project"

  This chapter describes how to plan an object-oriented development project.

- Chapter 4, "The Foreign Currency Exchange Application Project"

  This chapter provides an overview of the application and the residency project.

Part 2: Object-Oriented Application Development

- Chapter 5, "Object-Oriented Analysis and Design"

  This chapter describes general object-oriented analysis and design considerations and approaches.

- Chapter 6, "Modeling the Problem Domain"

  This chapter introduces the Visual Modeling Technique (VMT), a methodology that integrates visual programming with object modeling.

- Chapter 7, "Designing and Constructing the Solution"

  This chapter provides design approaches to be used with VisualAge.

- Chapter 8, "Sample Application: Design Work Products"

  This chapter describes the design work products of the application.

- Chapter 9, "Recommendations"

  This chapter provides practical recommendations for object-oriented application development with VisualAge.

- Appendix A

  This appendix presents the specifications of the sample application.

- Appendix B

  This appendix provides listings of the data definition language and REXX control files used to generate the sample application.

PREFACE.2 Related Publications

The publications listed in this section are considered particularly suitable for a more detailed discussion of the topics covered in this document.

- *Object Technology in Application Development*, GG24-4290-00

- *Client/Server Computing Application Design Guidelines: A Distributed Relational Data Perspective*, GG24-3727-00

- *Client/Server Computing Application Design Guidelines: A Transaction Processing Perspective*, GG24-3728-00

- *Client/Server Computing: The Design and Coding of a Business Application*, GG24-3899-00

- *VisualAge User's Guide and Reference*, SC34-4490

- *Construction from Parts Architecture: Building Parts for Fun and Profit*, SC34-4488-00

PREFACE.3 International Technical Support Organization Publications

A complete list of International Technical Support Organization publications, with a brief description of each, may be found in:

Bibliography of International Technical Support Organization Technical Bulletins, GG24-3070.

1.0 Part 1. Introduction

Subtopics

1.1 Chapter 1. The Visual Age of Application Development

Today's computing environment contains more challenges than ever. The typical information processing application requires services from multiple hardware platforms and several layers of both systems and application software. These application systems must also be integrated if organizations are to realize the full potential of their software, hardware, and personnel investments.

Constructing complex applications in this computing environment is a difficult task. Competition demands constant innovation. End users expect intuitive, easy to use, and robust business applications. These applications must be designed to be extensible, scalable, and responsive to changing business conditions. To meet these challenges, a combination of new and traditional technology is required.

Subtopics

1.1.1 Visual Programming

Over the last two years, a new software technology has matured enough to become a viable software development alternative. The technology is called *visual programming*, and it was developed in response to business demands for faster and more responsive application development. Visual programming promises to:

- Improve development processes

- Reduce the time required to deploy production applications

- Enhance applications by improving the interaction between developers and end users.

Visual programming enables nonexpert programmers to create whole or partial applications using a graphical approach rather than traditional textual languages.

Early visual programming technology focused largely on building user interfaces. It has recently improved. Robust tools are available that encompass both the data and process modeling components of application building.

Today many software vendors, ranging from the software giants to a group of small companies, provide visual programming products. A number of products in the marketplace provide different visual programming implementations. Some of these products are:

- Visual Basic** from Microsoft
- Visual C++** from Microsoft
- VXRexx** from WATCOM
- Powerbuilder** from Powersoft
- VisualWorks** from ParcPlace
- Enfin** from Easel
- PARTS** Workbench from Digitalk.

The newest trend in the visual programming development tools marketplace is to provide a complete application development environment that deploys both an object-oriented and client/server technology. VisualAge* from IBM* is one of the first products that provides such capabilities. It enables developers to enhance the existing business applications and their technology infrastructure with a new and powerful visual programming object-oriented technology.

1.1.2 VisualAge and Application Development

VisualAge is a client/server application development power tool. It focuses on business applications including both online transaction processing and decision support applications. VisualAge enables professional developers to quickly build the client portions of business applications, complete with a graphical user interface (GUI), application logic, and both local and remote resource access. VisualAge technology provides a pure object-oriented development environment, a set of interactive development tools, a library of prefabricated components, and a set of tools for client/server computing. All combine to provide a powerful application development environment. Figure 1 highlights the many features of VisualAge.

```
+------------------------------------------------------------------------+
¦                                                                        ¦
¦                                                                        ¦
¦                                                                        ¦
¦                                                                        ¦
¦ PICTURE 1                                                              ¦
¦                                                                        ¦
¦                                                                        ¦
¦                                                                        ¦
+------------------------------------------------------------------------+
```
Figure 1. A Dynamic VisualAge Programming Environment

Subtopics

1.1.2.1 Tools and Components

As noted above, VisualAge provides an environment that enables application developers to integrate existing applications with object-oriented technology. Here are some of the features of VisualAge:

Visual programming: VisualAge provides a visual programming tool that enables the creation of complete applications nonprocedurally using a highly productive approach called "construction from components." This approach consists of building applications using a repository of existing parts or building blocks. These parts can be distributed as components of a library provided with the system or built in-house.

Library of parts: VisualAge's prefabricated components include support for a GUI, database queries, stored procedures, transactions, communications, multimedia, and third generation languages (3GL) dynamic link library (DLL) access.

Graphical user interface: The GUI support included in the library of components enables the development of applications that conform to the Common User Access (CUA) specifications. The GUI also includes the extensions to support smart entry fields, tables, and forms.

Multimedia exploitation: Multimedia is the construction of animation, sound, video, and other media into interactive computer applications. IBM has created *Multimedia for VisualAge*, an addition to the VisualAge development environment, to help developers build applications that will take advantage of this new technology. With Multimedia for VisualAge, adding multimedia to traditional applications is easy.

Client/server and communications: VisualAge provides comprehensive support for client/server computing. This is made possible through multiple protocols and programming interfaces, such as:

- APPC (Advanced Program-to-Program Communications)
- TCP/IP (Transmission Control Protocol/Internet Protocol)
- NetBIOS (Network Basic Input Output Services)
- CICS OS/2 ECI (External Call Interface)
- EHLLAPI (Emulator High-Level Language Application Programming Interface).

Relational database support: VisualAge includes support for local and remote relational database access. This support is provided by VisualAge in the shape of visual programming components for SQL queries. These queries can be made dynamically through the VisualAge query builder or statically through the VisualAge 3GL DLL access facility. Currently supported relational databases include:

- IBM OS/2 DBM*, DB/2 2*, DB2* (through DDCS/2*), and DB2/6000*
- Oracle**
- Sybase**.

Enhanced dynamic link library support: This feature allows for direct execution of a DLL program from the VisualAge programming environment. An interface tool is provided that first creates the definitions that are required for a local C or COBOL DLL. These definitions include the necessary objects and methods required to execute in the object-oriented environment. VisualAge uses this feature and provides the generic DLL visual programming part. The DLL feature also provides full multithreading support.

Team programming: VisualAge exists in two editions:

- Personal Edition, an entry level product for individual programmers
- Team Edition, which provides support for team programming and configuration management.

VisualAge provides advanced and comprehensive support for team programming. A central library of components and classes are provided in a networked development environment, allowing a team of developers to collaboratively develop and manage software components.

Configuration management: VisualAge provides support for version control, release control, application relationship management, and subsystem packaging. This support is for both completed applications and

applications still in the process of development. VisualAge offers three types of relationship management:

- Parts relationship--A parts relationship arranges components into a hierarchy where each component is made up of many parts.

- Contains relationship--A contains relationship defines a component as existing within another component. It is similar to an aggregation or "part-of" relationship, but is more restrictive. The *contained* component cannot exist outside the *containing* component.

- Prerequisite relationship--A prerequisite relationship specifies that a particular component must exist in the VisualAge definition before a second component can exist.

1.1.2.2 VisualAge and Object Technology

VisualAge provides a pure object-oriented language, IBM Smalltalk, which can be used both to enhance and extend the applications that are created through visual programming.

Transition to object-oriented technology: The VisualAge product suite can facilitate a smooth transition to object technology. With VisualAge object technology can be introduced gradually in an enterprise and at a pace that is best suited for the organization. Some of the capabilities to assist users in this transition include:

- GUI creation capability
- Visual programming
- Extensive communications support
- The relational database interface.

A large immediate investment in Smalltalk and object-oriented skills is not required for *initial* VisualAge use. One can invoke logic that is already written in another programming language through DLLs. And, using both the DLL and networking interfaces, code written in most popular languages residing on host or server systems can be accessed and reused to build new applications. As an organization's VisualAge skill base improves, building applications with object-oriented methodologies and IBM Smalltalk is likely to be explored.

The VisualAge development environment: The VisualAge tool creates true object-oriented applications. The entire VisualAge development environment was created using IBM Smalltalk. IBM Smalltalk supports objects, encapsulation, inheritance, polymorphism, and model-view separation.

Through construction from components, VisualAge provides complete reuse of applications. Examples of reuse include GUIs, classes, methods, connections, and database interface specifications. As more proficiency is built with VisualAge, very sophisticated applications can be developed that are both reliable and extensible.

1.2 Chapter 2. Client/Server Computing and Object Technology

This chapter reviews the characteristics of Client/Server computing, defines the main terms used in this paradigm, and describes its relationship to object technology, illustrated by the definitions of the Object Management Group of the Common Object Request Broker Architecture (CORBA) and by SOM/DSOM, IBM's CORBA compliant product.

Subtopics

1.2.1 The Client/Server Computing Model

The Client/Server computing model for distributing applications defines a client application, for example, a PWS application program, which calls for services such as data or processing functions. These services are provided by a server that performs a function on behalf of the calling program (see Figure 2).

In this model, the programming complexities of distribution across the network may be handled by the called service or the calling mechanism. But the distinguishing characteristic of the Client/Server computing model is that the distribution complexities should be transparent to the client application, which is the calling program. The client program requests services by calling service routines that are available as part of the underlying distributed operating system or network operating system. The client application does not need to distinguish between local and remote services.

PICTURE 2

Figure 2. The Client/Server Computing Model

The sharing of resources within the workgroup and, if required, throughout the enterprise is accomplished through the interaction of requesting clients and supplying servers.

Client/Server computing makes it possible to develop business applications that provide flexible and adaptable business logic. The advantages of Client/Server computing are many. Foremost among the advantages is the enablement of:

- Hardware scalability
- Rapid change in business applications
- New technology for competitive business advantage.

The goal of the Client/Server computing model is to enable a client anywhere in a network to request services from anywhere else in the network in a transparent manner (location, function, performance, and vendor) and independent of any particular interconnection media. Additionally, a client system should be able to perform server functions at any point in time. To achieve this dual behavior, it is necessary to implement the network operating system using industry and international hardware, software, and communication standards. The current direction is to allow transparent communication among processors, as illustrated in Figure 3.

PICTURE 3

Figure 3. The Client/Server Computing Direction

The main concern with a generalized kind of distribution is related to performance. This problem must be addressed not only by improving the hardware and communication facilities, but also by providing software support functions for handling resource administration. For example, a more advanced form of interoperability, called intelligent interoperability, requires interaction among systems, some of which may have the capability of functioning as intelligent agents.

Distributed query processing is one example of intelligent interoperability: a global access plan is developed that attempts to optimize the use of individual database systems attached to a network. The intelligent interaction is achieved by an external global query optimizer component. Other capabilities that may be involved in intelligent interoperability include automatic invocation of required translation

between the interacting systems' languages and data structures, advanced forms of synchronization to coordinate access to shared resources, and systems that use knowledge of the cooperating activities to optimize overall performance.

One interim scenario, which is expected to be quite common on the road to full interoperability, is a model that comprises a local area network (LAN) operating system executing on client and server workstations. This system is called a workgroup LAN system (see Figure 4): a client PWS requests services from PWS-based servers by means of specific LAN formats and protocols.

PICTURE 4

Figure 4. Workgroup LAN System

The enterprise system, an evolutionary stage of the implementation of distributed systems, is shown in Figure 5. In an enterprise system, multiple workgroup LAN systems are supported by enterprise servers across the company. Therefore the model includes the integration of the LAN systems into the enterprise system. A communications approach by itself is not enough. It is necessary to develop a systems approach to achieve the desired transparency that full implementation of the Client/Server computing model offers.

PICTURE 5

Figure 5. Client/Server in an Enterprise System

At the heart of the Client/Server computing model is the concept of network computing, which has introduced a radically different approach to computing systems. This approach considers where the computer applications are developed and where they are executed. With proper design, changes made to the enterprise system (for example, adding another workstation, installing another software product for network management) should not affect the other users of the system.

A ***network computer*** is perceived as a single multiuser computing system that functions and performs like a traditional host computer but is built from a set of discrete machines, interconnected by a very high speed, highly reliable mechanism. This structure is transparent to the application.

1.2.2 Distributed Systems

In the field of distributed systems, it is often the case that the same terms are used with different meanings, usually because the context in which they are used is different. This section describes the meaning of some of these terms as they are used throughout this book.

Distributed processing is used as the all-encompassing or generic term. The other terms are considered subsets of the distributed processing concept; the concepts they cover may overlap, as illustrated in Figure 6. The diagram shows the computing approaches that can exist in an enterprise. On one side there is centralized host computing (for example, an online order entry system); on the other side there is stand-alone computing (for example, a spreadsheet application run by a business professional on a personal computer).

PICTURE 6

Figure 6. Enterprise Computing

Between the stand-alone computing and the centralized host approaches lies the whole spectrum of distributed processing. For instance, cooperative processing, a subset of distributed processing, is usually defined as a specialized computing paradigm where one of the distributed processors is a PWS. Cooperative processing applications may or may not be designed according to the Client/Server computing model. A peer-to-peer distributed application between a PWS and a host computer is an example of an application that does not fit the Client/Server computing model.

Client/Server computing is also a kind of distributed processing. Not all current distributed processing systems conform to the Client/Server computing model: in many cases, distribution of function and data is not transparent to the end user or the program. Transparency is a major criterion for an application that is designed according to the Client/Server computing model. Client/Server applications are also cooperative processing applications when one of the processors involved is a PWS.

The first implementations of distributed processing were designed to manipulate local data, and only the program logic was distributed. Distributed processing used to mean that the location of the data determined where the application processing was done, that is, the application function was moved to the place where the data was located. If the application needed remote data, techniques such as file transfer were used. This type of distribution does not involve interactive communication or data servers. Early distributed applications were implemented in a hierarchic way: the distributed processor had a "slave" relationship to the host, which was the "master."

Applications can be made to simulate a Client/Server type interaction with remote request tools such as the Enhanced Connectivity Facility (ECF). In this way, the distribution is made transparent to the end user, but not to the physical designer.

With the evolution of the technology, distributed processing acquired more capabilities. Nowadays a broader definition for distributed processing is accepted: any application in which processing takes place across two or more linked systems is called distributed; the data needed for the application can also be spread across the interconnected systems.

Cooperative processing and Client/Server computing are special cases of distributed processing. Often they overlap.

Subtopics

1.2.2.1 Cooperative Processing

Cooperative processing is a type of distributed processing where the resources required to complete an application are split between two or more processing units. The cooperative applications sometimes present a peer-to-peer relationship between a client application program running on a PWS and another application program running on any platform providing the required services. Although the workgroup data and the application logic commonly reside on the server, and the PWS is usually used for presentation services and private data, other types of processing distribution are not excluded. In most cases, the server machine is the host computer.

The cooperative processing model uses the PWS as a front-end for applications that customarily run on enterprise servers. The application developer writes the multi-user part of the application to run on the host or server, and the single user part on the PWS; in the peer-to-peer version of the cooperative processing model, there is a need to design and implement explicit function distribution across computing platforms (see Figure 7).

```
+--------------------------------------------------------------------------+
¦                                                                          ¦
¦                                                                          ¦
¦                                                                          ¦
¦                                PICTURE 7                                 ¦
¦                                                                          ¦
¦                                                                          ¦
¦                                                                          ¦
¦                                                                          ¦
+--------------------------------------------------------------------------+
```
Figure 7. Peer-to-Peer Cooperative Processing

1.2.2.2 Client/Server Computing

Client/Server computing is accomplished through a logical relationship between requesting clients and responding servers. The model allows a client, that is, an application usually (but not necessarily) residing on a PWS, to access in a transparent manner services provided by one or more servers that can be running on another PWS on the LAN, or on a mainframe computer.

An example of Client/Server computing is the workgroup LAN, which is characterized by LAN media, network-operating-system-level function, and application execution taking place primarily on the client workstation.

Access to distributed services is said to be transparent to the client application program when the application contains only business logic. Other logic, such as for presentation, retrieving data, and printing, is provided by client and server elements of the network-operating-system that will interact by means of a defined communication mechanism (see Figure 8).

```
+------------------------------------------------------------------------+
¦                                                                        ¦
¦                                                                        ¦
¦                                                                        ¦
¦                                                                        ¦
¦                                PICTURE 8                               ¦
¦                                                                        ¦
¦                                                                        ¦
¦                                                                        ¦
¦                                                                        ¦
+------------------------------------------------------------------------+
```

Figure 8. Client/Server Application Model

In summary, the Client/Server computing model enables a group of computers connected by a low latency network to be transformed into a multiuser system, which, from an application perspective, is used and programmed much like any conventional midrange or mainframe system. When network speed and reliability approach the internal speed of a midrange or mainframe computer, the network operating system transforms the network into a conventional multiuser system.

1.2.3 Client/Server Technology Requirements

The concept of a network operating system, as required for the implementation of the Client/Server computing model, has been around for quite a while, and it has been implemented experimentally on many occasions. Its practical application, however, depends on the availability of a low-latency, highly reliable communications path between the client and the server.

Subtopics

1.2.3.1 Low Latency

In the design of a conventional operating system, a great deal of attention is paid to the average time the system requires to perform services on behalf of its application programs (response time), as this has a major effect on the overall performance of the application. Such fundamental system services include basic I/O, task, process, and memory management.

The considerations regarding response time are not different with network operating systems. As a general rule, using a network operating system is only practical if the overhead imposed by the network itself, that is, the communication path, is negligible in comparison to the latency of a given service when executed in a nondistributed manner. In other words, applications should run on a network operating system no more slowly nor less reliably than on a conventional operating system.

Some current network operating system products may actually be faster than their nondistributed counterparts. Access to files on the attached server may very well be faster than access to files on a local client fixed disk, because the server machine can provide more buffer memory for cached data than could be supplied on a client workstation. The server may also allow the client to access a faster disk than the disk in the client's hardware. This illustrates a key characteristic that has made the Client/Server computing model an industry requirement. The model enables workstations to share common resources--for example, file cache, faster and larger direct access storage devices (DASDs)--and therefore helps to eliminate the need to add these potentially expensive resources to each workstation. So in addition to providing access to shared data and programs, the Client/Server computing model is also viewed as a cost-effective means of expanding the resources of individual workstations.

Usually, latency is considered to be imposed mainly by the raw speed of a network segment. But when looking at a network from a Client/Server computing perspective, the end-to-end latency must be considered, which involves several factors:

- The speed of the slowest network segment between the client and the server

  Because network speeds have been optimized for geographic locality, the Client/Server computing model is mostly applicable today within a single building or campus.

- The number of bridged segments between the client and the server

  When LAN systems grow, they are linked together by bridges and routers. As geographic restrictions are reduced, the number of bridged LANs increases. Bridges impose a certain level of propagation delay. Routers, which interconnect different types of networks, can impose an even greater delay.

- The latency of the attachment mechanisms on the client and server platforms

  Although personal computers provide a comparatively direct path from the network interface to main memory and the main processor, this is not true for midrange and mainframe systems, largely because of historical downward compatibility constraints. Those systems were designed to optimize overall throughput for a large number of concurrent and lengthy I/O operations. Lengthy refers to the actual elapsed time and is thus a function of both the communication speed and the number of bits of data. Midrange and mainframe systems are architected to accommodate a comparatively high average data length to channel speed ratio. The optimum communication design for a network operating system assumes a comparatively low average data length to channel speed ratio.

  The Client/Server computing model is characterized instead by a short request and response interaction, and the current midrange and mainframe I/O design imposes significant handshaking.

- The communication protocols

  Communication protocols originally designed for wide area network (WAN) computing were responsible for ensuring efficient and reliable delivery of data between the programs at either end. This involved locating the target, finding the optimum route when multiple choices were available, sending multiple segments of a large set of data concurrently (even on multiple routes) with reassembly at the receiver, detecting and acknowledging errors and retransmitting just those segments, providing secure access and a variety of other features. Although this type of communication is very useful for sending large amounts of data through an untrusted and potentially

error-prone network, the assumptions made are inconsistent with the requirements of the Client/Server computing model. Even as higher speed WANs become more generally available, they may still display comparatively higher error rates and have more potential security exposures than geographically limited networks.

Technology continues to evolve, and new products are constantly introduced with new design points. Exactly what the geographic range of a network operating system can be, and which products can participate in it, is a constantly changing equation. What is essential to implement the Client/Server computing model is a viable technical approach that maximizes the insulation between the products that are part of the network operating system and those that interoperate with it.

1.2.3.2 Response Time and Usability

Although low latency is certainly a critical factor in the success of Client/Server computing, the transparency offered by this model offers a level of usability that has value in and of itself. As a result, the Client/Server computing model is often desirable even when some degree of latency must be introduced. Uniform access to any data in the network, for example, is desirable, and users may decide to accept some response time penalty in order to have the convenience of uniformity.

1.2.3.3 Openness

A network of discrete machines is inherently an open environment because each machine represents a complete and separately obtainable product. In fact, this concept becomes a major advantage in the Client/Server computing model. Because a network operating system is not a single piece of code, any system platform that can be made to support the required network formats and protocols can be linked together to become a component of a network operating system. Thus a network operating system, unlike a conventional operating system, is not limited to supporting a single software or even hardware platform.

Because each of the machines and products that work together as a network is separately obtainable, there is no need for the users of a networked multiuser system to buy the same product, to purchase it at the same time, or to work in the same fashion. This has particular appeal in environments where all end users do not perform the same job on the same data. In such environments, each user may have a different task but needs to work with others to complete a job. The workgroup has been the primary marketplace for Client/Server computing so far, but the benefit of the flexibility of Client/Server computing is more generally applicable than just to the workgroup environment.

Client/Server computing allows for flexible, granular growth of a system. Additional functions, platforms, and processes may be added to a network system with significantly less effort than is typical with a single multiuser system.

1.2.4 Implementation of Client/Server Computing

Distributed systems can be implemented through facilities provided by the database manager, the transaction manager, or through different communication mechanisms such as message queueing, remote procedure calls, and conversations.

Subtopics

1.2.4.1 Client/Server Computing with Database Managers

The Client/Server computing model, as implemented with the database manager, requires that all data access and transfer be done by the database manager and the network operating system, without the application or user having to worry about how the access and transfer take place. This subject is treated extensively in the subsequent chapters of this book; it is briefly discussed below.

Distributed Relational Database Architecture (DRDA) defines service protocols extending the SAA Common Communications Support (CCS) in an architected manner, providing the ability to access and use distributed relational data throughout the enterprise, for all database management systems that have implemented it.

DRDA allows transparency between the application and the distributed relational databases by providing the formats and protocols required for distributed relational database management. It provides three functions and two protocols. The functions are:

- Application requester (AR) functions
- Application server (AS) functions
- Database server (DS) functions.

The application requests service for data from the relational database management system using the SQL API through the AR. The resolution of the routing of the request through the network of databases is handled by the AR, AS, and DS functions. These three basic functions, provided by DRDA, are linked by two connection protocols:

- **Application support protocol**, which provides the connection between ARs and ASs.

  The AR supports the application end of the DRDA connection by making requests to the AS, and the AS supports the DBMS end by answering these requests.

- **Database support protocol**, which provides the connection between ASs and DSs.

The flow of these functions and supporting protocols is illustrated in Figure 9.

```
+----------------------------------------------------------------------+
|                                                                      |
|                                                                      |
|                                                                      |
|                                                                      |
|                               PICTURE 9                              |
|                                                                      |
|                                                                      |
|                                                                      |
+----------------------------------------------------------------------+
```
Figure 9. DRDA Network. Functions and Protocols

1.2.4.2 Client/Server Computing with Transaction Managers

Client/Server computing can be implemented using the functions or facilities provided by IBM's transaction managers. For example, CICS (1) provides the following functions, whose use depends on which Client/Server application type is being implemented:

- Function shipping
- Distributed program link (DPL; CICS OS/2 only)
- Transaction routing.

(1) Throughout this section the generic term CICS represents CICS OS/2 and all CICS host products, except for CICS/VM.

Subtopics

1.2.4.2.1 Function Shipping

Function shipping allows a CICS application program to issue a request for CICS services that is processed by another CICS system. For example, a CICS application may issue a file read request, which is processed on the remote CICS system that actually owns the file. The *shipping* of the read *function* is totally transparent to the application program. This transparency allows for the implementation of applications conforming to the Client/Server computing model. Function shipping gives CICS users the ability to access data resources on another CICS system. These resources can include VSAM files, BDAM files, IMS databases, and CICS transient data and temporary storage.

A CICS OS/2 application can access data owned by a CICS host system, and a CICS host application can access data owned by a CICS OS/2 system, provided that in each case the resources are defined as remote in the function shipping system. The resource's location can be predefined to enable the transaction to access it transparently. Additionally, the resource's location can be changed. The CICS systems involved in function shipping may reside in different processors at different sites or in different regions of the same processor.

Notes:

1. A CICS application cannot function ship SQL requests; DB2 handles this type of function shipping.

2. CICS OS/2 cannot function ship requests for DL/I or DB2 databases. The distributed program link function (see below) needs to be used to access DL/I databases. (Distributed transaction processing can also be used.)

1.2.4.2.2 Distributed Program Link

DPL allows a program running under CICS OS/2 to issue a CICS LINK command, which is essentially a CALL to another program running under the control of another CICS system, either CICS OS/2 or host CICS. This is, effectively, function shipping of the CICS LINK function. It is currently not possible for a host CICS program to use DPL to link to a CICS OS/2 program.

DPL provides an easy way of accessing DL/I and SQL databases on a host CICS system. It also provides the possibility of improved performance as compared to function shipping; for example, a single DPL can achieve a data set browse that would require multiple flows if function shipping were used.

1.2.4.2.3 Transaction Routing

Both CICS and IMS provide transaction routing services. IMS supports transaction routing through its Multiple Systems Coupling (MSC) facility and its Intersystem Communication (ISC) facility. Both facilities provide connectivity between IMS systems, allowing the use of distributed transactions and the sharing of data.

MSC can connect two or more IMS systems. ISC can connect one IMS system with other IMS systems, or with CICS systems or user-written applications. Both MSC and ISC can be used to connect systems in the same or in different machines.

The ISC facility enables application designers to distribute transaction requests among IMS, CICS, and user-written applications. Users can thus route transactions that access data on remote systems. This distribution is transparent to users, as long as the transactions have been previously defined as remote.

1.2.4.3 Client/Server Computing with Communication Mechanisms

To build distributed systems according to the Client/Server computing model, the following communication mechanisms can be used:

- **Messaging**

  This mechanism stems from the online transaction processing (OLTP) environment. Programs can formulate work requests as messages, at least in the respect that a requester can drive parallel servers by dispatching messages to several servers and wait for completion messages from all of them, rather than calling each one serially. The asynchronous transmission of messages, or packets, as a queueing and scheduling facility is central to this model.

- **Remote Procedure Call (RPC)**

  RPC is a mechanism underlying some of the distributed systems in the UNIX environment. With this mechanism, services are requested through subroutine calls to procedures. This structuring of an application into sets of services and users of the services is familiar to most programmers. A call and return interface allows remote functions or services to be invoked as if they were local.

- **Conversation**

  With the conversation mechanism, a program requests creation of an instance of a partner program, and the program and its partner exchange information over the conversation. Once created, the relationship between the partners is peer-to-peer; both can send and receive. This relationship is often considered a synchronous communication model and can support only one logical unit of work (UOW) at a time. An example of this model would be any LU6.2 conversation. Although not natively Client/Server, an RPC-like communication can be simulated with the conversation mechanism.

1.2.5 Client/Server Applications

Client/Server applications can be categorized on the basis of such characteristics as the location where the application is executed, the reason for execution at that location, the fundamental purpose of the application, and the level of transparency to users and developers.

There are three basic application types:

- Client-based applications

  As the name implies, even though the code may be stored on a server, client-based applications execute on a client platform. Client-based applications are currently popular because there are so many off-the-shelf PWS applications that either will be, or are already, adopted for LAN use. Because of the very good transparency of LANs to the application, most applications require no modification at all to exploit file server capabilities because they are single-user applications that assume private access to data. Client-based applications may also require access to shared data, but the sharing is a facility of the server, not a characteristic of the application.

  Developers of client-based applications will simply assume that general sets of "system" capabilities are available for general application use. These sets of services for client-based applications are sometimes supersets of services available on the client.

- Server-based applications

  Server-based applications are most similar to the traditional centralized time-sharing and transaction processing applications. They are executed on the server, because they either are existing applications or there is an advantage to being local to service resources, for example, computing power to satisfy high transaction throughput and information bandwidth for data or devices.

  Server-based applications are usually multiuser and require the added complexity of managing the use of application resources by many users.

- Network-based applications

  There are two fundamental types of network-based applications, each of which exploits the respective capabilities of its client and server execution platform:

  - Cooperative applications

    Cooperative applications distribute application function across two or more platforms within the network, either local or wide area. Portions of the application uniquely identify with each other across the network. This allows each platform to do what it does best. For example, a cooperative application may exploit the GUI capabilities of the PWS, while accessing a powerful multiuser set of functions on the workgroup or enterprise server. The application developer is specifically aware of the functionality available from each platform and must specifically construct the application accordingly. This may be desirable when generalized services are not available yet or may never be generally available because of their specialized nature.

  - Distributed applications

    Distributed applications also execute on multiple platforms within the network; however, processes within a distributed application are generally available on the network and can be dynamically distributed to the most appropriate and available platform for execution. For example, a numerically intensive process within an application could be migrated from a client to the network's computing server, while a complex database query, within that same application, may be migrated to the data server. This is the application model favored by the Open Software Foundation's Distributed Computing Environment (OSF's DCE).

Subtopics

1.2.5.1 Object Technology and the Client/Server Computing Styles

The Gartner Group has adopted a model for Client/Server computing (see Figure 10). The model has been readily adopted by the information processing industry.

Many of today's object-oriented implementations follow the "remote data management" computing style of the Gartner Client/Server model. This style also is the most popular style of implementation. Remote relational databases are a critical piece of Client/Server systems because they provide the data required for business application support for many organizations.

```
+------------------------------------------------------------------------+
¦                                                                        ¦
¦                                                                        ¦
¦                                                                        ¦
¦                                                                        ¦
¦ PICTURE 10                                                             ¦
¦                                                                        ¦
¦                                                                        ¦
¦                                                                        ¦
+------------------------------------------------------------------------+
```
Figure 10. Client/Server Computing Styles (Gartner Group)

1.2.5.2 Distributed Object Computing

Distributed object computing (DOC) represents a new and exciting trend to merge two powerful technologies in the information industry, namely, Client/Server computing and object-oriented technology.

DOC will allow the application developer to assemble applications from objects that run on disparate platforms distributed in a network. Objects communicate with each other through a message-passing mechanism. An object's role can change between both client and server. A given object may act as a server to some objects and, at the same time, as a client of other objects. Furthermore, these client and server objects may be on the same machine and even in the same business process. DOC means that the distribution of objects across the network is possible but not necessarily mandatory. This architecture holds significant promise for tomorrow's distributed applications.

<u>DOC and Client/Server Computing</u>: Emerging standards for object-to-object communication, such as the Object Management Group's (OMG's) Common Object Requester/Broker Architecture (CORBA), may facilitate the delivery of off-the-shelf objects for distributed systems. The CORBA specification defines the interfaces for sending messages from one object to another. This common standard should assist application designers in object-oriented application integration over multiple platforms and operating system environments.

1.2.5.3 VisualAge and IBM's System Object Model

The first release of VisualAge will support IBM's Systems Object Model (SOM*), which provides component reusability and interoperability across languages.

IBM's SOM and DSOM implements CORBA: IBM's SOMobjects technologies, including the SOM and Distributed Systems Object Model (DSOM), were designed to enable the creation of industrial strength, binary object classes that are truly reusable and are scalable to client server configurations. SOM is a language-neutral object model that allows developers to package object classes in such a way as to provide an enhanced ability to reuse, modify, and customize them within and among different language compilers. SOM, initially used in the development of the OS/2 Workplace Shell, will become a core technology for "packaging" object frameworks that are designed to be extended by others, such as the forthcoming OpenDoc compound document architecture from IBM, Apple, Novell, and Wordperfect. SOM has the unique capability to separate the object's type from its implementation, resulting in a very flexible and dynamic model for developing object-oriented applications. DSOM, a scalable extension of SOM, provides transparent, distributed object services that are fully compliant with the OMG's CORBA specification.

Expected benefits from VisualAge SOM support: By integrating the strengths of the VisualAge development environment with IBM's SOM and DSOM technology, VisualAge developers, and the end users of the resulting applications, will benefit from the following enhancements to the VisualAge development environment:

- Cross language object classes

  Using currently available development tools, object classes developed in one language environment cannot be effectively reused in another language environment.

  The VisualAge SOMsupport will allow VisualAge developers to reuse and subclass object classes developed in other languages, as well as allow VisualAge Smalltalk object classes to be reused by other languages (if they also support SOM). This possibility of cross language reuse of classes can significantly increase the availability of object classes to be used in VisualAge as well as allow VisualAge classes to be used effectively in other development environments (such as C or C++). In all, the support of SOM by VisualAge will allow developers to reuse more code, thus resulting in less costly and risky development.

- Support for CORBA distributed objects

  Client/Server computing is quickly becoming a normal requirement for the development and deployment of new applications. The OMG's CORBA standard (as fully implemented in IBM's SOM/DSOM technology) was designed to provide a productive, flexible, and dynamic Client/Server solution by exploiting the power of distributed object services. IBM's DSOM is a scalable extension of SOM that provides local and remote distributed object services across a heterogeneous network.

  VisualAge SOMsupport is planned to support the DSOM extensions and the ability to develop CORBA-based distributed objects (workgroup enabling code is required for deployment). VisualAge SOMsupport can significantly decrease the costs and risk associated with developing, deploying, and adapting Client/Server applications while enhancing the migration path from local objects to distributed objects.

- Support for SOM-based operating system services

  Over time, an increasing number of operating systems services will be packaged as SOM object classes and frameworks. The first of these frameworks was the OS/2 Workplace Shell, which allows developers to customize, extend and integrate into the Workplace Shell desktop as well as inherit behavior and characteristics (such as drag/drop) when developing new applications. OpenDoc, a compound document architecture supported by IBM, Apple, Novell, WordPerfect, and others, will be packaged as a framework of SOM classes that can be used in developing document-centric applications.

  VisualAge SOMsupport is positioned to exploit these frameworks that will help automate the design and development of a new wave of document-centric, collaborative applications in a very productive and high quality fashion.

- Industry standard class definitions

The SOM interface definition language fully conforms to the OMG's CORBA Interface Definition Language (IDL) standard as does the SOM interface repository (providing run-time access to class information and definitions).

The VisualAge SOMsupport Smalltalk bindings will evolve to be compliant with OMG standards as they are made available. VisualAge SOMsupport classes will, therefore, adhere to industry standards (CORBA IDL), allowing customers to retain investments in skills and code, while providing a mechanism for interoperability with other CORBA-compliant object request brokers (such as Hewlett Packard's HP ORB).

IBM's SOM is not intended to replace existing object-oriented languages (such as Smalltalk). Rather, it is intended to complement them so that application programs written in different languages and with different compilers can share common SOM class libraries that are CORBA compliant. SOM also provides a rich set of object-oriented characteristics that can complement existing object-oriented languages (such as Smalltalk and C++) as well as procedural languages (such as C and COBOL). In summary, SOM, when integrated with the VisualAge development environment, can result in a more productive and powerful set of application development capabilities.

1.3 Chapter 3. Planning an Object-Oriented Development Project

Delivering an application in a timely manner is important. Of equal or greater importance is delivering the *correct* application, an application that meets all of the user requirements. The definition of *correctness*, however, usually changes as the project progresses and the requirements are modified. The development process must be able to accommodate these changes. Of utmost importance, the project plan must recognize and allow for these changes throughout the application development lifecycle.

Subtopics

1.3.1 The Iterative Process Model

The waterfall process model has been used for application development projects for several decades. With this model each project phase is completed serially. The completion of each phase is a prerequisite for the start of the next. Because the output from previous project phases tends to be frozen, there is not enough flexibility for the application development environment.

There are many advantages to delivering rapidly an early version of the application, gathering user feedback, producing a refined version, gathering feedback again, and so on. This process model, called *iterative*, allows for a better compliance with the user requirements. The model is not privy to object-oriented application development, but is particularly well suited to the characteristics of object-orientation.

The next section describes in detail the iterative process.

1.3.2 Iteration: Plan, Produce, and Assess

Each iteration of the development of an application consists of planning, producing, and assessing phases. Because we learn from each iteration, the scope of the iterations changes over time. The scope of each activity during each iteration varies depending on both the size of the project and the iteration objectives. Figure 11 shows how planning, producing, and assessing change in each project iteration [LOR91].

```
+------------------------------------------------------------------------+
¦                                                                        ¦
¦                                                                        ¦
¦                                                                        ¦
¦                                                                        ¦
¦ PICTURE 11                                                             ¦
¦                                                                        ¦
¦                                                                        ¦
¦                                                                        ¦
+------------------------------------------------------------------------+
```
Figure 11. Planning, Producing, and Assessing

1.3.3 Advantages of the Iterative Process

Iterative development techniques are not new. The basic premises that underlie iterative systems development are to:

- Develop the main functions of the system
- Revisit the analysis phase to refine, improve, and add function.

Business executives, project managers, methodologists, and developers have examined past project failures and realized that:

- We understand simple things before we understand complex things.
- We build systems poorly before we build them well.

The iterative approach allows progress in stages. We can learn the basic concepts and system requirements and then add to this understanding as we progress through the lifecycle. At the end of each iteration, the result can be verified by end users. Even at an early stage in development, prototypes can be delivered and end users can add or amend requirements. This way, requirements do not have to be fully specified at the beginning of the project. Instead, they are dynamically identified and refined. The application under development can easily incorporate new and better understood requirements. The resulting product is more likely to be a valid solution that addresses the needs of users.

Through iterative application analysis we refine our application knowledge and build the optimal solution. This iterative process provides significant advantages over waterfall process techniques. It allows us to continually refine the:

- Initial requirements
- User interface
- Application model
- Application function.

1.3.4 Building a Basic Plan

The easiest approach to building a project plan is to define both the project and the intended result. Once a basic definition is complete, we can decompose the end product to determine its components.

We approach the object-oriented banking application in a very similar fashion. We have a basic objective to complete. We build a plan to deliver an end product. The end product is an application. The application is comprised of a series of iterations that address basic requirements, analysis, design, code, and implementation. The composition of the iterations is described below.

A Sample Iteration: Our project is divided into three application iterations. See Figure 12 to view sample project tasks and estimates for the first iteration.

PICTURE 12

Figure 12. Sample Iteration I

1.3.5 Object-Oriented Application Development Teams

Building systems with a new planning technique and new technology requires accurate task estimation. However, with a new technology, project planners cannot rely on the estimates from inexperienced application designers and developers. If project expectations are to be met, the composition and skill base of the development and planning teams are key to the success of the project.

1.4 Chapter 4. The Foreign Currency Exchange Application Project

The project run at the ITSO-San Jose Center had the following goals:

- Define a methodology to build real-life object-oriented applications with VisualAge.

- Apply the proposed methodology to build an object-oriented banking application with VisualAge. The application was designed to run in a Client/Server systems environment.

Object-oriented analysis (OOA) and object-oriented design (OOD) techniques were used to build the banking application model. The application was implemented with VisualAge. This document presents the application development process using OOA and OOD techniques and the VisualAge product.

Subtopics

1.4.1 Objectives

The project objectives were to:

- Detail the business application's problem domain and build a cohesive object-oriented solution

- Apply object-oriented systems development methodologies in a Client/Server environment

- Explore the linkage between the object model and the VisualAge application implementation

- Explore the project-relevant features and functions of VisualAge and Team Programming

- Examine the construction of the sample application from component parts

- Document the result of the experience and recommend VisualAge application development guidelines.

1.4.2 The Application

Selecting an Application: We have selected a Foreign Currency Exchange (FCE) application for our project. A similar application has been described in the ITSO publication, *Client/Server Computing: The Design and Coding of a Business Application*, GG24-3899, using a traditional analysis and design approach.

Requirement Specifications: The application chosen provides foreign currency exchange services. The exchange services provide foreign cash and foreign travelers checks to customers based on current exchange rates.

The bank has a series of worldwide branches. These branches perform the foreign currency exchange services. Customers that have an account in a bank branch are considered bank customers. Customers that do not have an account in any bank branches are considered nonbank customers. Both bank customers and nonbank customers are eligible to use the exchange services.

The FCE application provides the branch cashiers with country information, for example, country currency name, denominations of both cash and travelers checks, and foreign country currency restrictions.

The cashiers get customer information, create an order, and receive an immediate response regarding the requested stock availability in their drawers and in the local branch. Customer orders can be pending, in process or completed. Orders only become pending if sufficient stock is not available in the cashiers' drawer or in the local branch.

The system manages foreign currencies, customers, orders and payments, and stock availability.

Please refer to appendix A for a complete requirement specification of the sample application.

1.4.3 Application Iterations

The project team agreed to have three project iterations. The first iteration was focused on:

- Understanding the problem domain
- Identifying and defining candidate objects
- Modeling the relationships among objects
- Building a limited function prototype.

Again, the primary focus was on understanding the problem. More time was spent in planning than in producing.

The second iteration used the knowledge gained from the first iteration and focused on:

- Refining an object model and assigning responsibilities to objects
- Looking for class structures and potential reuse
- Building an expanded function prototype
- Looking at systems architecture solutions.

Some time was spent in planning, but we started to produce a design prototype. Object models, use cases, preliminary object responsibilities, and a limited function prototype were the expected output from this iteration.

The third iteration used the knowledge gained from the first two iterations and focused on:

- Designing the object model
- Designing application class structures
- Designing the application
- Designing a systems architecture.

2.0 Part 2. Object-Oriented Application Development

Subtopics

2.1 Chapter 5. Object-Oriented Analysis and Design

Analysis can be defined as the process of describing the problem to be solved (the answer to the *what* question, design as the process of describing the solution (the global answer to the *how* question), and programming as the process of implementing the solution (the detailed answer to the *how* question).

The most recent advances in object technology have been in the areas of analysis and design. Many practitioners have advocated that the object-oriented approach moves much of the software development effort into analysis and design. Some would even argue that object technology differs from the traditional approach fundamentally in its modeling techniques.

However, many programmers using object-oriented languages who work independently or in a small team are known to prefer to write code almost from the start of the application development process, relying heavily on prototyping to capture user requirements and then evolve the prototype to a final solution. This approach is flawed, however; often a prototype coded in this way does not scale up the production environment requirements. It is therefore necessary to define two types of prototypes: one that is used to capture the user requirements, and another that will be the base for the development of the final product.

In this chapter, we explore the roles of object-oriented analysis and design. Following some commonly used object-oriented methodologies, our attempt is to find a practical approach to object-oriented development, using VisualAge as both the prototyping tool and the implementation tool.

Subtopics

2.1.1 Object-Oriented Analysis Considerations

Analysis is an important step, because it helps us to attack the right problem. Without proper analysis, we may build an elegant solution but end up solving the wrong problem.

The analysis phase thus has two main objectives: understanding the problem domain and defining the desired application behavior.

Understanding the problem domain means that during the analysis phase the analyst should gain enough knowledge of the characteristics of the environment and elements that are related to the problem the application is supposed to solve, and of the scope of the application as defined by the user expectations, before starting any production activity.

Defining (modeling) what the application is all about (as opposed to how it should carry out the desired behavior) means that decisions concerning the final concrete implementation of the application should be delayed as much as possible to a design phase.

How much is "enough knowledge" of the problem and how much "degree of freedom" is affordable during the analysis phase depends on many external factors, such as user availability, project schedule, and environment constraints and therefore is subject to the judgment of the software engineer. However, large-scale projects, such as nationwide complex transaction processing systems, will still need thorough analysis and design to ensure that complex, multidivision business requirements are met.

Subtopics

2.1.1.1 Object-Oriented Modeling

The analyst should be aware of the great differences between traditional (functional) and object-oriented modeling approaches.

Traditional Approaches: Traditional structured analysis and design techniques enforce a peculiar view of what a software system is: a software system is a collection of functions (processes) with side effects on stored data. As a consequence, the desired application behavior is defined by the events (inputs) to which the system will react, the processes that produce the desired output from a given input, the elementary subprocesses in which each process can be decomposed, and the effects that processes have on stored data. Data modeling is mainly used as a support technique to obtain a correct design of stored data.

Object-Oriented Approach: Object technology changes the view we have of what a software system is: a software system is a collection of objects that interact with each other to provide the required services. As a consequence, the desired application behavior is defined by:

- The events to which the system reacts
- The objects involved in servicing all external events
- The message flows and interactions among objects.

The objectives of object-oriented analysis are to:

- Find the "right" objects for the application

- Abstract the objects to classes

- Model the relationships between classes (This describes the knowledge each object must have of other objects in order to interact with them.)

- Define which services each object belonging to a certain class must provide

- Model how a given external event is serviced by object interactions.

Finding the Right Objects: An object-oriented application consists of objects requesting services to each other to provide a required application behavior.

How can we find the right objects that represent the application? How do we find the classes that represent common meaningful traits of these objects?

This definition seems to be accepted: good objects are objects that provide an elegant and extensible partition of the application's problem domain (see :bibref refid=tay93.). Good objects, that is, objects meaningful to the domain model, are the result of a thorough understanding of the problem domain and of its available computer based solutions.

Different methodologies present diverse approaches to the problem. For instance, many authors [RUM90], [BOO91] suggest that a first group of candidate objects can be obtained by performing a noun analysis on the problem statement. The set of model objects is completed when building the use cases that describe the application. In other cases [JAC92], [GOL92], the starting set of objects is derived from the description of the interaction between the user and the system. The description of the interactions is called "use cases," "scripts," or "scenarios," depending on the author.

Objects can be abstracted to classes that are like templates or "cookie cutters" that describe the common traits or characteristics of the objects meaningful for the application. The abstraction of objects into classes, as well as the abstraction of classes into superclasses, depends on the context in which this abstraction is considered. For instance, depending on the context of our model, an albatross can be clasified as an animal or as a flying object.

Once the classes are identified, the modeling activities can then be carried out using the classes as the modeling constructs. Those activities include finding relationships among classes and defining their attributes and methods.

Modeling Relationships: There is a difference in the relationships between objects in an *object model* and the relationships between entities in a *data model* (see :bibref refid=jac92., page 173).

In data modeling we are interested in finding abstract static relationships between entities (for example, Cashier works_for Branch), whereas in object modeling (behavioral modeling) we are interested in finding the *roles* that objects play in their relationship with other objects (the Branch is the "employer" for a Cashier, and the Cashier is the "employee" of the Branch).

Relationships are bidirectional in entity-relationship modeling, but not in object modeling [Jacob92]. The characteristic of the associations to be unidirectional is denoted by the names of the roles the objects play in their associations. For example, it could be relevant for the behavior of a person to know about the president of the US, but not vice versa.

This consideration lead to some design and implementation issues: For example, let's say there is some kind of relationship between objectA and objectB; then we could have:

- ObjectA knows about objectB but not vice versa: in this case objectA is responsible for maintaining the relationship.
- ObjectA knows about objectB and objectB knows about objectA: in this case we have two pieces of information that *must* be kept in sync and we must assign, at design time, the responsibility of maintaining the relationship either to objectA *or* to objectB.

2.1.1.2 Modeling the User Interface

The purpose of user interface modeling is to gain sufficient knowledge to ensure that the user interface design is meeting the needs of a user.

User interface modeling is a complex process that requires domain knowledge as well as user interface and design skills. It is important that the user interface designer change his or her perspective to that of the user who will use the user interface to get the work done. The modeling and design work should reflect an emphasis on the user's model (how the interaction with the computer maximizes the user's productivity) and downplay the developer's view (use of resources and computer performance). Needless to say that a delicate balance between the two is important. A design with a poor user interface is not a correct solution for the user.

A major activity in the user interface modeling process is to select a common user interface style that will capitalize on the experience the users already have and fits into an experience framework that most users will recognize. This allows a user to immediately recognize familiar objects and work with them and hence minimize training requirements. Common user interface styles, such as the desktop and forms metaphors, are well documented and accepted by users and developers.

The other decision that we have to make in the modeling process is to select the design approach or framework. Two frameworks are well documented:

1. Model-View-Controller (MVC) - An object is modeled into three pieces, which are themselves objects:

    - A *model* accesses and stores data on request.

    - A *view* presents information to a user.

    - A *controller* controls the behavior of the object based on user input. The benefit of designing objects in this way is better reuse of objects. Common views can be developed that use a variety of models. Common models can service a variety of views.

2. Model-View/Controller (MV/C) - This is a simplification of the MVC framework where the controller and the view are combined into a view component because of complex dependencies between the two. Although reuse of these combined objects is reduced, the design of controller and view components together is less complex task. Many implementations use this framework.

Regardless of whether the designer decides to use the MVC or the MV/C framework, the key design consideration is a model-view separation, which encourages the separation of business and presentation logic. In this way, changes to either logic will not affect the views.

An example of this type of separation is data entered by a user that is evaluated and returned to the user. Presentation logic would contain the collection and return of data. The business logic would contain the evaluation rules and process. In this way, the user interface could change to collect the data through a different media without changing the evaluation and processing of the data. In the same manner, the evaluation and processing of the data could change without changing the presentation of the data. Another benefit of this separation are the distribution possibilities in a Client/Server environment. The presentation logic resides on a workstation where speed of performance is necessary for a GUI. The business logic can reside elsewhere and be shared by several other machines.

Regardless of implementation, it is useful to start designing with reuse in mind and compromise reuse with simplicity when hitting implementation limits.

User interface design is an iterative process. We iterate through the development of a GUI prototype until the user is satisfied. This can be done in two stages: when the analyst tries to understand the user functional requirements and when the final production GUI is developed.

Figure 13 shows a high-level application design framework that reflects this model/view separation approach. It also includes the data aspects of the application design that have to be considered for any real-life application.

```
¦ PICTURE 13                                                              ¦
¦                                                                         ¦
¦                                                                         ¦
¦                                                                         ¦
+-------------------------------------------------------------------------+
```

Figure 13. Application Design Framework

2.1.2 Object-Oriented Design Considerations

The object-oriented analysis activities are aimed at building an analysis model of the application satisfying user behavioral requirements, that is, a model of:

- The objects needed to provide the required application behavior
- The relationships between objects, that is, the knowledge each object must have of other objects in order to be able to interact with them
- The services each object must provide
- How external events are serviced by object interactions.

The purpose of design, on the other hand, is focused on the solution regarding *how* to:

- Map domain requirements into computational architecture
- Complete the definition of the relationships found in analysis by specifying the responsibilities or roles of each class in the relationship
- Reflect the structure of objects from both the problem domain and solution domain
- Support application behavior and goals
- Specify hardware and software constraints in implementing the solution
- Make tradeoffs between reusability, modifiability, and efficiency.

A practical implementation of an analysis model must also take into consideration the nonbehavioral requirements, such as response time, availability, data integrity, and environment constraints (on hardware and software configurations).

Two main design activities can be devised: *system design* and *object design*.

System design produces a high level *system architecture*, which defines the hardware and software configurations of the target environment, and an *application architecture*, which defines the partitioning of the system into subsystems and the allocation of subsystems to the target processes. The processes can be allocated to the same processor or different processors according to the system design considerations.

Object design activity refines and extends analysis models in order to allow a practical and efficient implementation for the target environment.

Subtopics

2.1.2.1 Design for a Client/Server Environment

Designing for a Client/Server environment, from a software engineering point of view, means that the application is partitioned into components that interact with each other using a protocol that is transparent to the underlying communication and transport mechanism: a component (the client) asks for a service and another (the server) component provides the service. Only the client has intelligence of the final objective of the service required (so only the client has knowledge of the logical UOW).

Designing for a Client/Server environment, from a systems engineering point of view, means that the underpinnings to support the application must be designed to enable a client to ask for the service of a server no matter where it is located in a computer network, providing for complete location transparency. Appropriate "middle layer" components must be chosen to provide all of the required security, routing, and presentation services that this transparency concept implies. (2)

Client/Server System Design Considerations: A Client/Server solution is inherently complex. The designer must deal with many system design considerations, such as:

- Function and data placement
- Local and remote transparency
- Networking and connectivity
- Data access and data integrity
- Security
- Performance
- Scalability.

A discussion of Client/Server system design issues is outside the scope of our work (which focuses on application design). The reader can refer to the ITSC redbooks [TKA91a], [TKA91b], and :bibref refid=STA92., for an indepth discussion of the matter.

However, we do offer the following design considerations, which are relevant to VisualAge in a Client/Server environment:

- VisualAge supports the main distributed computing styles implemented in various Client/Server enabling technologies currently available:

  - Remote databases (through SQL access to multiple database types, including DB2/2*, DB2/6000*, Oracle**, and Sybas**e)
  - Distributed data (through DRDA*)
  - Distributed functions or cooperative processing (through CICS* ECI, APPC*, TCP/IP*, NETBios*)
  - Distributed presentation (through EHLLAPI*).

  So VisualAge does not constrain system design alternatives.

- These technologies are not designed to support object-oriented concepts. In fact, they are intended for the distribution of data or functions, not objects, so the design must accommodate the effort to bridge the gap between an object-oriented application design and a non-object-oriented system architecture.

- A great deal of design simplification and productivity gain can be expected with the advent of emerging technologies that provide object-oriented system support.

Client/Server Object Design Considerations: The object-oriented design approach fits naturally in a Client/Server design prospective. In fact, object interaction is a Client/Server interaction by its very nature. For example, object A sends a message to object B asking for a service, object B performs the service. This interaction is provided by IBM SOM/DSOM* and is available for many environments, including VisualAge. However, some aspects of the technology require careful design consideration:

- Currently available technology does not provide efficient direct *persistence* support for Smalltalk objects. And as VisualAge parts are essentially Smalltalk objects, they live in the user's personal image which is in virtual memory during run time. The correct persistent storage support must be chosen for the application, be it a database or flat file; and the Smalltalk objects must be mapped to persistent storage structures, such as table rows or records.

- Currently available Client/Server technology does not provide any level of *location transparency* to Smalltalk objects. Smalltalk objects cannot send messages to objects residing in another image. Therefore, a traditional (non-object-oriented) Client/Server enabling support must be chosen for problems involving distributed database or distributed functions.

- Finally, *object placement and/or replication* on different images must be designed to address data consistency and integrity.

  For example, in our FCE application, to complete his or her work, the cashier needs to interact with a CashierDrawer object to verify currency availability, and with a Currency object to obtain the exchange rate. There are many design decisions to make. For instance, will both of these objects be present in the cashier's image or will the cashier interact remotely with them? Will these objects be replicated in some other cashier's image? (In fact, all of the cashiers need to interact concurrently with the Currency objects.) How can we maintain consistency between different replicas of the same object?

The design challenge is to merge the two worlds (the object-oriented model and the non-object-oriented Client/Server enabling technology), keeping the best of both (the conciseness and robustness of the object-oriented model and the services provided by the Client/Server enablers).

(2) The term *middleware* is often used to refer to this middle layer, which is between the application and the network operating system that provides the application and network services.

2.1.3 Visual Programming Considerations

In this section we discuss the following topics:

- The general scope of visual programming
- What can be done visually with VisualAge?
- If both alternatives are available, that is, if the same program logic can be implemented visually or nonvisually (programatically), which is the better alternative?

General Scope of Visual Programming: Smalltalk applications generally follow a structuring or partitioning scheme according to the *Model-View-Controller (MVC)* paradigm.

- The model is the set of objects actually implementing the application's business logic (an Order, a Stock ...).
- The view represents the user interface which consists of a set of objects that interact with the user (a list box, a push button).
- The controller is the set of objects that transform the user interaction with the view into actual requests to the model.

According to this application partitioning scheme, a visual programming tool can be used to:

- Visually define the user interface objects; the developer imbeds these objects in a traditional programming environment
- Visually define the user interface objects and the control objects; the developer then builds the business objects in a traditional programming language, but all interactions to or from the user interface are specified visually.
- Visually build the entire application.

What Can Be Done Visually with VisualAge: Although VisualAge does not explicitly show the MVC separation, it does provide full support for the creation of *view* objects with the underlying *controller* objects. In fact, the interactions between widgets and the *model* objects are fully specified by the arrowed connections in the visual part through the composition editor.

The next step to a full visual application development should be to build nonvisual parts from components (by dropping selected parts into a composition editor and specifying visually their interactions).

When building nonvisual objects, two situations can be considered:

- Building composite nonvisual parts that have only simple aggregation behavior. That is, the composite part only acts as a container for other parts. For example, a box containing candies may not have any interesting behavior to allow the user to access its content.

- Building composite nonvisual parts that have a more complex aggregation behavior. That is, the composite part provides its services, which may be delegated to its subparts. For example, a PS/2 is not just a box containing a power supply and a mother-board. Rather, it provides the user with a switch: when the switch is turned on, the "switch on" action is actually delegated to the power supply as a "power on" request.

Building Simple Aggregation Parts: This can be done easily with VisualAge and with effective results by dropping the selected parts into the composition editor and adding them to the part external interface.

Building More Complex Aggregation Parts: It is questionable whether visually building a complex logic part is easier than just doing Smalltalk programming; with the current display technology the screen becomes rapidly cluttered with parts and connections and the whole meaning of the picture becomes obscure.

It is feasible, but the following limits have to be circumvented:

- Actions cannot be easily delegated to subparts, because VisualAge does not allow an Action (external) to Action (internal) connection. This means that a Smalltalk method is always needed to service an action. This method must raise an event that is in turn connected to the internal action.

- Events cannot be easily passed from a subpart to the "external."

- "General logic" parts (condition testing, looping) are not available.

- Collection access functions are not available (visual equivalent of the select:, detect:, and collect: Smalltalk iterators).

2.1.4 Selecting a Methodology to Work with VisualAge

A software engineering methodology is a process for the organized production of software, using a collection of predefined techniques and notational conventions. A methodology is usually presented as a series of steps, with techniques and notation associated with each step :bibref refid=rum91..

Subtopics

2.1.4.1 The Need for Object-Oriented Methodologies

A methodology is an essential part of object-oriented development--it outlines the philosophy and prescribes the way in which the analysis, design, and implementation will be derived.

Although there is no perfect methodology, as each methodology has its strengths and weaknesses, following a methodology is better than having none at all. The lack of a methodology could lead to a chaotic situation where the project is without a single, consistent philosophy or a roadmap it needs to succeed. For object technology to be accepted in mainstream application development, the use of a methodology is crucial.

2.1.4.2 Which Object-Oriented Methodology to Use?

Development could be done following one of the well-known object-oriented methodologies advocated in many popular textbooks, such as :bibref refid=rum91., :bibref refid=wir90., :bibref refid=boo94., and :bibref refid=jac92..

Our opinion is that all of these methodologies have their strengths and provide a range of applicability. Together they provide a powerful set of tools and techniques for object-oriented analysis and design.

It is interesting to note that, despite their apparent dissimilarities, most object-oriented methodologies have more commonality than differences in their approaches to modeling. However, it is not practical to use a "blended" approach in which we mix and match the best parts of some well-known object-oriented methodologies. It is difficult to reconcile the terminology and semantic differences among the various methodologies as well as the different notations they use.

Even though many object-oriented CASE tools support multiple notations, very few support the conversion from one notation to another.

A practical approach is therefore to choose one methodology, such as OMT (see below), as the "backbone" methodology to follow and stick to its notation throughout the entire development lifecycle and use the techniques from other methodologies to support the modeling and development effort. We call this a complementary approach.

Let's take a quick look at some methodologies that are commonly in use today, before we discuss why a complementary approach is desirable.

<u>Object Modeling Technique (OMT)</u>: OMT, developed by J. Rumbaugh et al., is a nearly comprehensive methodology that consists of several phases:

- Analysis

  The analysis phase concentrates on the understanding and modeling of the application and the domain within which it operates. The analysis results in a "generic" definition of the system. Objects, relationships, event flows, and functions are detailed. These are represented in the three types of models: object, dynamic, and functional. Together, they provide three complementary views of a system.

  - The object model captures the static view of what is the problem. The object model represents the static object structure, which shows the objects in the system, relationships between objects, and the attributes and operations that characterize each class of object. OMT suggests that the object model can be built from the elements of the problem statement: nouns are potential objects and verbs are potential associations.
  - The dynamic model represents the temporal, behavioral, and control aspects of the system, including the sequences of events, states, and operations that occur within the system of objects. The OMT uses scenarios to describe interaction sequences between objects. Events contracted and created by objects are stored in event diagrams. The dynamic model is represented graphically with state diagrams. Each state diagram shows the state and event sequences permitted in a system for an object class.
  - The functional model specifies the meaning of the operations of the objects as defined by actions in the dynamic model. The functional model is represented with data flow diagrams.

- Design

  There are two phases of design: *system design* and *object design*. System design results in an implementation structure for the system. Subsystems are defined and allocated to. Object design results in a detailed definition of the system. The analysis model is elaborated on, mapping the problem (business) domain to the solution (computer) domain.

OMT can be considered as a static methodology :bibref refid=tka94. because data structure is emphasized more than function.

<u>Responsibility-Driven Design (RDD)</u>: The responsibility-driven design, developed by Rebecca Wirfs-Brock et al., is a dynamic methodology that emphasizes the encapsulation of object behavior and structure. RDD is a dynamic methodology because it specifies object behavior before object structure and other considerations are determined.

RDD is an anthropomorphic approach requiring the analyst to think of

objects as collaborating agents with responsibilities. The collaborating objects assume the role of either a client or a server. A client requests the server object to perform services; and a server provides a set of services based upon requests. The requests a server can support and the client can request are defined and grouped into *contracts* (3). And contracts are specified in terms of *protocols* for each operation of a service. *Subsystems* are introduced to group classes to provide higher levels of functional abstraction.

The RDD modeling process includes two phases:

- Exploratory phase

  The exploratory phase achieves three goals--finding the classes, determining responsibilities, and identifying collaborations--through the following steps:

  1. Read the specifications
  2. Walk through scenarios. Write design cards.
  3. Create class cards. Look for nouns in the requirement specifications.
  4. Define responsibilities. Look for services for the clients of a class. Look at is_a (specialization), has_a (attributes), and is_analogous_to relationships.
  5. Identify collaborators: Look for class-to-class collaboration to fulfill the responsibilities identified to define the architectural interfaces between classes and subsystems.

- Analysis phase

  The analysis phase involves refining the object behavior and service definitions specified in the exploratory phase. The refinement includes defining interfaces (protocols) and constructing implementation specifications for each class. The work involved includes class refinement and specifications, class polymorphism definitions, and service specifications. The analysis phase produces:

  - Hierarchy graph
    To place classes in the inheritance hierarchy
  - Abstractions
    Look for abstract classes, to facilitate sharing common services and attributes.
  - Contracts
    Look for logical groupings of responsibilities
  - Subsystems
    Group tightly coupled classes into subsystems.
  - Collaboration graphs
    Show class relationships and identifies collaborations within and between subsystems
  - Protocols - specifications for message formats
    Provide textual supplemental information on subsystem, contract, and class cards.

Wirfs-Brock's RDD approach applies the technique of the CRC cards to identify class responsibilities and their collaborators.

The practice of use-case analysis was first formalized by Jacobson :bibref refid=jac92.. Jacobson defines a use case as "a particular form or pattern or example of usage, a scenario that begins with some user of the system initiating some transaction or sequence of interrelated events." A use case represents a dialog between a user and the system.

Use cases fit naturally into the scheme of object-oriented analysis and design because they portray user interactions with real-world (problem-domain) objects.

<u>The Need for a Complementary Approach</u>: Although OMT provides a nearly comprehensive methodology and notation and is one of the most popular methodologies in use today, it has some shortcomings that can be compensated for by adopting techniques from other methodologies.

OMT suggests the use of scenarios including the concept of an *actor*, but it is not as formalized as *use cases* from OOSE. We elected to apply use-case analysis to complete the identification of the user requirements, the activities the user performs, and the services the system must provide for the user. This approach will lead to a better understanding of the interactions between the user and the system to be built.

OMT suggests identifying object classes from the problem specifications. This is a reasonable starting point. Our analysis process is later completed by developing the use cases, and picking from the use cases potential object classes.

The justification for this approach is that we expect to better capture the user requirements by focusing both on the static and the dynamic aspects of the problem domain.

While Rumbaugh focuses more on the importance of associations (relationships) between classes, Wirfs-Brock's methodology focuses on the responsibilities of objects rather than their attributes.

RDD uses the CRC techniques which we found quite practical in fostering group dynamics among the (small) team of software designers :bibref refid=bec93.. In our experience, it is quite effective to use CRC to identify object responsibilities and collaborations after constructing the OMT object model.

We can also apply the concept of *subsystems* from RDD to map to either *applications* or *subapplications* as used in the VisualAge team programming environment. The mapping to subapplications offers a logical fit but presents reusability problems. The recommended approach of the IBM Cary Lab is to map subsystems to applications.

We decided not to follow through the RDD method to construct collaboration graphs and contracts between subsystems, mainly because without an automated CASE tool, drawing collaboration graphs manually is usually a messy and tedious task.

Jacobson's use of three different object types--entity objects, interface objects, and control objects--also provides some interesting ideas to map the object model to the MVC aspects of an application design framework. (4)

(3) Rebecca Wirfs-Brock has dropped *contracts* from the latest version of her methodology. However, we find that there is a need to formalize the protocols of the interaction among objects

(4) Wirfs-Brock calls these object types "stereotypes."

2.1.4.3 Prototyping with VisualAge during Analysis and Design

Object-oriented prototyping is an effective approach to testing that the user's requirements are being met by the system under construction.

VisualAge is a visual programming tool with rich supporting framework classes. In addition to its role as an implementation (programming) tool, it can be used as a prototyping tool with a modeling methodology.

Analysis Prototyping: In this document, we use the term *analysis prototyping* to mean developing the initial user interfaces (views) of the system to be built. The intent is to provide a proof-of-concept analysis prototype to solicit or validate the user requirements. Object-oriented modeling and analysis prototyping are important means to understanding the problem domain and the needs of the end users.

Both OMT and OOSE recognize the importance of user interface prototyping during analysis.

"It is hard to evaluate a user interface without actually testing it. Often the interface can be mocked up so that users can try it. Decoupling application logic from the user interface allows a 'look and feel' of the user interface to be evaluated while the application is under development." :bibref refid=rum91. (5)

"To support the use case model, it is often appropriate to develop interfaces of the use cases. Here a prototype of the user interface is a perfect tool. In this way, we can simulate the use cases for the users by showing the user the views that she or he will see when executing the use case in the system to be built." :bibref refid=Jac92.

Design Prototyping: Design prototyping, on the other hand, is to construct working prototypes to validate the feasibility of the design for the system being built. The premise is to build incrementally, gradually adding funtionality. Each version of the prototype shows a partial solution of the system, until the prototype has evolved to a final solution.

Using VisualAge as the design prototyping tool fits well with the "construction from components" paradigm. With VisualAge, new applications or subapplications can be assembled from reusable components that already exist. These components can be either visual (view) parts or nonvisual (application logic or model) parts. VisualAge thus achieves Rapid Application Development (RAD) by making it quick and easy to assemble, customize, or change a working prototype.

(5) However, this requires the definition of a "semantic interface" relating the meaning of the view and model objects.

2.2 Chapter 6. Modeling the Problem Domain

In this chapter, we take a closer look at the activities involved during the analysis stage. We present the steps of our approaches to model the business problem at hand. Wherever possible, we explain the rationale behind the approaches taken, exemplify the tasks and techniques in each step, and document the sample analysis work results.

Subtopics

2.2.1 Visual Modeling Technique (VMT): A Complementary Approach

As described in "Which Object-Oriented Methodology to Use?" in topic 2.1.4.2, we constructed a methodology covering the life of the application, from the requirements-gathering phase to the application testing phase. This methodology uses several techniques derived from existing modeling methodologies, and it is geared toward the use of a visual programming tool, such as VisualAge.

Subtopics

2.2.1.1 Justification for Creating VMT

There are many good object-oriented methodologies, and it would seem hard to justify the creation of a new one. However, when integrating visual programming in the application development process, some special requirements are added to the application development life cycle.

It has been noted [Tka94] that none of the published methodologies integrates explicitly in the modeling process either the GUI building or the prototyping phase. These activities are key for reaping the benefits of a visual programming tool. Therefore, in order to provide a systematic way of building object-oriented applications with VisualAge, we had to create a new approach. This approach was based, however, on existing and proven techniques and was inspired by the comments of the authors of the methodologies about their shortcomings and what elements from the other methodologies they would like to add to their own. These comments were made mostly at a memorable "Shootout at the OO Corral" session at the OOPSLA 93 conference, held in Washington, DC during October 1993.

Following that reasoning, we selected Object Modeling Technique (OMT) as our "back-bone" methodology and used its notations throughout. However, while OMT works best starting with a problem statement such as that provided by a request for proposal (RFP), we felt that it did not provide a mechanism to capture user interactions from the beginning. The scenarios in OMT are used for dynamic modeling, once the relevant objects have been identified from the problem statement. On the other hand, the use cases requirement modeling technique from Object-Oriented Software Engineering (OOSE) has the advantage of capturing the many modes of user interaction and is particularly suited for formalizing the results obtained by prototyping. A set of reusable GUI prototypes is also a byproduct of the prototyping activity.

The mechanism used to identify relevant objects in use-case analysis is similar to the one used in OMT to analyze the problem statement: both are based on language-syntax-related precedures, such as *noun analysis*. In building our domain object model we found it particularly productive to use the union of the classes and relationships found in both use-case analysis and problem statement analysis, because those classes usually represent different, albeit complementary, aspects of the application domain.

Once classes and relationships are found, there is a need to establish the distribution of responsibilities among the classes. We applied to that end Rebecca Wirfs-Brock's Responsibility Driven Design (RDD) approach to define object class responsibilities and collaborations, using the CRC cards as an aid in the process. We did not use CRC cards for finding classes. Although this may be useful in a teaching environment, we believe that CRC cards should be better used in class design, for distributing class responsibilities in the model. Eventually, new collaborations may become apparent when class responsibilities are changed. The soundness of this approach was confirmed in conversations with Kent Beck, one of the creators of the CRC cards.

The next step was to find and define the required class methods. Event trace diagrams and state diagrams (OMT) are used in this stage to model the changes of states and interactions of objects. The event traces define the messages among objects, and therefore, the methods to be invoked. The changes of states help us to understand the variables (attributes) affected by the methods.

The effect of the methods was defined by preconditions and postconditions, which were related to events as follows: given certain preconditions, an event can change the state of an object; the method invoked to produce this change of state should leave the object in a new state such that the defined postconditions are met.

Figure 14 depicts a high-level scheme of the VMT complementary approach. The following are the main analysis activities during the analysis stage in developing the FCE application:

- Start with a set of requirement specifications (problem statement).
- Develop use cases from the required services.
- Build requirement model from use cases.
- Develop analysis prototype using VisualAge.
- Find candidate objects from problem statement and use cases.
- Develop first cut of our object model. (We used OMTool for documenting the object model.)
- Define the class responsibilities and collaborations using the CRC technique
- Perform dynamic modeling, create event trace diagrams and state transition diagrams.
- Iterate and refine the object model.

+--+

PICTURE 14

Figure 14. The VMT Complementary Approach to Object-Oriented Analysis and Design

2.2.1.2 Requirements Modeling

Many analysis methodologies start the analysis work from a well-specified set of "user requirements." However, the "requirements document" does not always portray a clear picture of all requirements or user expectations.

As said before, we believe that a better understanding of user expectations and an easier transition from requirement to analysis can be achieved by building a more sophisticated model of the requirements.

The approach described here is exemplified in "Use Cases" in topic 2.2.2.1.

<ins>Actors and Use Cases</ins>: The steps in transforming a requirement specification to a requirement model are:

1. Find actors.
2. Develop use cases.
3. Represent use case relationships.

An *actor* is any entity outside the system with which it interacts. Actors--as the name implies--are active entities, their behavior is not determinably predefined. Rather, they are the source of events to which the system must react.

A *use case* is a sequence of interactions between an actor and the system aimed at obtaining some application service.

Actors are typically user roles, such as the cashier, the branch manager, or external (client) systems. Finding actors helps us to identify *what is inside* and *what is outside* the system; that is, to identify the system boundaries. (Finding actors is a process somewhat similar to the process of building a context diagram in Structured Analysis; see :bibref refid=You89..)

Finding the use cases helps us to identify how the system will typically be used by actors. Conversely, the services that must be provided define the use cases. *Use cases identify the services that must be provided.*

A very first high-level model of the system consists of actors and use-case support (see Figure 15).

PICTURE 15

Figure 15. First Use Case Model

<ins>The Requirement Model</ins>: A well-defined set of relationships can be drawn between use cases. Use case B *extends* use case A if it can be inserted in use case A and thus extends the description of use-case A. This allows for changes and additions of the use case function.

Use case A *uses* use case B, if a sequence of actor-system interactions needed to complete use case A is described in use case B.

The problem domain may be represented in term of actors, use cases, and use-case relationships (see Figure 16).

PICTURE 16

Figure 16. Use Case Model

<ins>Detailing Use Cases</ins>: Uses cases can be described in many different levels of detail. For example, we could say:

1. The cashier creates a new order
2. The cashier enters the order details
3. ......

Or:

1. The cashier logs on to his or her workstation
2. The cashier selects New from the menu
3. The cashier enters the customer name
4. ......

The correct level of detail depends on an optimum balance between two contrasting goals: gathering enough information (which calls for a more detailed description) and not to commit too early to a specific solution (which calls for a higher-level description).

Our suggestion is to proceed in a *top-down use-case refinement*, sketching initially just a very high-level picture of the services provided and then extending them with more significant variants.

2.2.1.3 Analysis Prototyping with VisualAge

The software engineer and the target users of the application should jointly develop use cases to reach a common understanding and a common set of vocabulary for the problem domain.

VisualAge has powerful prototyping capabilities that can be leveraged to assist in requirement modeling.

During analysis prototyping, we developed the initial user interfaces (views) of the system to be built. The intent was to develop a proof-of-concept of the user requirements.

2.2.1.4 Finding Objects

We elected to start identifying the potential object classes after we developed the initial use cases. The rationale was this: although the problem statement describes the static aspects of the application (that is, the data-centric aspects), the use cases provide a better understanding of the interactions between the potential user and the system to be built and give some insight into the behavior that the application must provide. Both aspects--structural (static) and behavioral (dynamic)-- are complementary.

Objects are the building blocks we use to describe the problem domain and to build the application. Objects account for the information held in the system and for its global behavior.

Not everything is an object. Objects have two distinct characteristics:

- They have a state; that is, they maintain some kind of information.

- They exhibit behavior; that is, they provide some kind of service.

A useful heuristic for finding objects is to perform a *syntactic analysis* on both the use cases and the problem statement.

Nouns or noun phrases are candidate objects, verbs are candidate services provided by objects, and adjectives identify possible different kinds of the same object (subclasses).

The human language is quite ambiguous; therefore some further analysis (*pruning*) is required:

- Two nouns can refer to the same thing (synonyms), so a single name must be found.

- Nouns can refer to something that is not relevant to the problem.

- Nouns can refer to something that can be better modeled as an attribute: for example, the noun *amount* in the sentence "the total amount of cash held in acashier drawer" can be better modeled as an attribute of cashier drawer.

- Nouns can refer to something that can be better modeled as a service: for example, the noun *withdrawal* in the sentence "the cashier makes a withdrawal from the customer account" can be better modeled as "withdraw" service provided by the account.

The software engineer and the target users of the application should jointly discuss the syntactic analysis results as this activity will solve many of the ambiguities of the use cases.

There is a vast literature on the strategies to use to find objects (nearly all books on object-oriented analysis listed in the bibliography present some discussion of the subject).

2.2.1.5 Object-Oriented Modeling

The main objective of analysis is to deliver a complete description of what the application does. Two different aspects must be addressed: the *static structure* and the *dynamic behavior*. Two models capture these two aspects:

- The static object structure is described in an *object model*.

- The application dynamic behavior is described in a *dynamic model*.

The two models are tightly connected, as they show two different aspects of the same application. In fact, some integrity rules can be applied to check that they are congruent and consistent with each other. Therefore, the process of building the two models is *not sequential*; the two models have to be developed together in an *iterative* fashion.

Subtopics

2.2.1.5.1 Object Modeling

The object model provides a description of the application object structure, including:

- The object classes that make up the application

- The associations between objects

- The information that each object maintains (its attributes)

- The services that each object provides.

The object model is the primary source of information for the result of the analysis work and the starting point for the design and implementation effort.

We adopted the OMT notation (see Figure 18 on page 56) for the description of the object model. OMT notation is rich and allows for a comprehensible representation of many object-oriented concepts. OMT notation is in essence an extension of the extended Entity-Relationship notation, which includes inheritance relationships.

```
+------------------------------------------------------------------------+
¦                                                                        ¦
¦                                                                        ¦
¦                                                                        ¦
¦                                                                        ¦
¦ PICTURE 17                                                             ¦
¦                                                                        ¦
¦                                                                        ¦
¦                                                                        ¦
+------------------------------------------------------------------------+
```
Figure 17. OMT Notation

2.2.1.5.2 Detailing the Object Model

During analysis, the object model should be used to represent business objects, any "implementation object" (wrappers, devices .....) should not be present. In other words, the object model should capture the "essential" structure of the application domain objects, including:

- The attributes that actually define the object status, but not, for example, the derived attributes.

- The services that actually define the object responsibilities, that is, the object interface, but not, for example, the algoritm of the methods or functions that implement these services.

The inheritance hierarchy should not be detailed early during analysis unless a very clear difference in behavior is apparent between different subclasses of a class, because this level of detail can limit the flexibility for the follow-on design work.

Dynamic Modeling: "The dynamic model describes those aspects of a system concerned with time and the sequencing of operations - events that mark changes, sequence of events, states that define the context for events, and the organization of events and states." :bibref refid=rum91.

The dynamic model provides a description of the application dynamic behavior showing:

- The flow of events in the application: *event traces*
- How the state of each object is modified by the flow of events: *object state diagrams*.

The dynamic model is represented graphically with state (transition) and event trace diagrams. Each state diagram shows the state changes and event sequences permitted in a system for one class of objects, while the event diagram shows the interactions among objects and with the external actors.

"A single state transition diagram represents a view of the dynamic model of a single class or of the entire system." :bibref refid=boo94.

Not every class has significant event-related behavior, and so we supply state diagrams only for those classes that exhibit significant event-related behavior of the system as a whole. During analysis, we use state diagrams to indicate the dynamic behavior of an individual class or of collaborations of classes.

Event Trace: An event trace (Figure 19) provides good insight into an application's behavior and valuable information for deciding the services that each object should provide.

Events are any communication of information to an object; in our environment "communication of information" means:

- The user requests some service
  or
- An object requests some service from another object
  or
- An object returns information on the completion of the requested service to the client object or the user.

In an event trace diagram, the objects requesting and/or providing services are shown as vertical bars, events are shown as directed segments from the client object to the server, the sequence of events is shown proceeding from the top of the page.

```
+--------------------------------------------------------------------------+
¦                                                                          ¦
¦                                                                          ¦
¦                                                                          ¦
¦                                                                          ¦
¦ PICTURE 18                                                               ¦
¦                                                                          ¦
¦                                                                          ¦
¦                                                                          ¦
+--------------------------------------------------------------------------+
```
Figure 18. Sample Event Trace

The sample event trace in Figure 18 shows the following suite of events:

1. The user requests some service from the system; the service is actually provided by Object1.
2. Object1, in order to fulfill the requested service, in turn requests

some service from Object2 (Object2 *collaborates* with Object1 to provide the user-required service).
3. Object2 returns the requested service to object1.
4. Object1 returns the user requested service to the user (system response).

Note that the flow of events at the system boundary is actually the scenario described in the use cases.

The event trace does not show any internal details of objects, but only their role in the application (the events that they service and the events they produce). So, for example, the work that object1 undertakes before sending its service request to object2 is not shown.

The set of events serviced (received) by an object represents its "responsibility" in the system.

Sometimes it is obvious that a service request has a corresponding response event from the server object, and it is not necessary to show it. This is valid only if the mechanism of service request from a client to a server has a synchronous (call-like) semantic. In fact, if message passing is asynchronous, object1 continues its work after having sent the service request to object2 and must be able to intercept the completion event.

<u>Detailing the Event Trace</u>: An event trace can be drawn in many levels of detail (from very high-level service requests down to the actual Smalltalk message passing).

We used event traces to explore and define the high level object responsibilities, that is, to define the *role* an object plays in the system.

2.2.1.5.3 State Diagrams

Object behavior can change over time in response to events. For example, an order just created accepts modifications from the user, but an order accepted does not accept the modifications.

Usually we can account for the differences in object behavior by introducing the concept of a state. The state of an object is defined by the values of its attributes or internal variables. For example, an order just created is in a "draft" state, and an order accepted is in an "accepted" state, where "draft" and "accepted" are possible values of the attribute *status*.

Object state diagrams are shown with a bubble representing each state of the object. Inside the state, the activities that the object performs while it is in the state (and the services it requests other objects to perform) can be represented. A labeled arrow (transitions) exiting from a state shows the events accepted (the requests that can be serviced) while the object is in the state; the label is the name of the event accepted. The arrow points to another state if servicing the request implies a status change. The arrow points to the same state if servicing the request does not imply a state change.

An object state diagram can be derived from event traces:

1. Choose an object in the event trace (it is represented by a vertical bar).
2. Represent (or locate) on the state diagram the state in which the object is supposed to be when this event trace starts.
3. Follow the vertical bar and look at incoming events:
   - If the object changes its state in response to the incoming event, represent the new state on the diagram and draw an arrow from the current state to the new state.
   - If the object does not change its state in response to the incoming event, draw an arrow from the current state to itself.

   The transition can be labeled with the name of the incoming event.

The dynamic model of an application is a collection of state diagrams that interact with each other through shared events. Figure 24 on page 76 shows the first creation of a state diagram from an event trace, and Figure 24 in topic 2.2.2.5 shows the integration of a second event trace in the same state diagram.

We emphasize the use of event trace diagrams. We did not rely heavily on state diagrams, because our business application objects did not show very complex dynamic behavior.

<u>Functional Model</u>: OMT includes functional modeling, which models the functions performed by the system using data flow diagrams (DFDs). DFDs are useful for showing function decomposition and can be quite helpful when complex calculations are involved.

DFDs do not play a central role in object analysis; however, they can be used to describe the algorithms for the methods.

2.2.1.6 Responsibility Analysis

"...finding objects isn't the hard part of design; the hard part is distributing behaviors among objects." :bibref refid=bec93.

We believe that a "responsibility driven" analysis is a good way to tackle the problem of determining what the objects are supposed to do, which in turn defines their public interfaces. Therefore, such analysis can be considered part of the design activities.

Responsibility analysis is based on two concepts:

- An object has certain *responsibilities* in the system, such as:
  - Maintaining some kind of knowledge
  - Providing some kind of services to other objects.
- An object provides requested services by *collaborating* with some other object.

Modeling an object's responsibilities and collaborators helps to define the distribution of attributes, services, and associations among the classes. For instance, given an association linking two classes, we define who does what in this association, which determines the placement of attributes and methods.

This approach suggests a high-level anthropomorphic view of object interactions, which is quite healthy during application analysis.

Class responsibilities can be derived from use cases. To that end, we defined the system responsibilities implied by the use cases, and then we assigned responsibilities to the classes in the system.

An example can help to understand the process. Let's examine the following use case: A cashier populates an order adding the requested currencies. If some currency is not available in the stock, the system notifies the cashier. A first-cut object model of the objects involved in this scenario is shown in Figure 19.

PICTURE 19

Figure 19. The First-Cut Object Model for the Foreign Currency Exchange Management System

This simple scenario highlights some system responsibilities:

- Knowing the currencies ordered in a order and allowing the user to modify them,
- Knowing the availability of currency in a stock, checking the availability of currency in the stock etc, and so forth.

Here "to know" means to be able to retrieve and modify the value of an attribute.

We then assign responsibilities:

- Order should know about ordered currencies.
- Order should be able to accept currency requests.
- Stock should know about currency availability.
- Order should check currency availability before accepting a currency request.

Collaborators are then defined for each object responsibility, by verifying whether the object has all of the information needed to fulfill its responsibility. If this is not the case, collaborators must exist to provide the required information.

In our example we find that Order does not know about currency availability and must somehow collaborate with Stock in order to get the information. In the object model (refer again to Figure 19) we can see that Order cannot collaborate with Stock because no relationship exists between them. This forces us to perform a more in-depth analysis: we

could discover a new relationship (showing the fact that the order must be fulfilled from a stock) or extend the collaboration pattern (for example, the order could ask the cashier which stock is to be used and then ask Stock to fulfill the order).

In summary, the information gathered during responsibility analysis allows us to:

- Define object services

- Refine attributes

- Verify and refine relationships.

A useful technique for responsibility analysis is CRC. CRC uses index cards where, for each class, responsibilities and collaborators are described. CRC cards have proven to be a useful development tool that facilitates brainstorming and enhances communication among developers. The technique was first proposed by Beck and Cunningham as a tool for teaching object-oriented programming :bibref refid=bec89.. Refer to :bibref refid=bec93. for additional information about responsibility assignment.

2.2.1.7 Preparing a Data Dictionary

A data dictionary includes precise descriptions for each class. These descriptions help us to understand the purpose and definition of the class. Attributes, operations, and associations for the class are also included.

2.2.1.8 Iterate and Refine the Object Model

Object-oriented analysis may require many iterations before the analysis is complete. For example, in our small application, we completed several iterations of OOA and several corresponding iterations of database design activities. Each design activity added more details to the database entity-relationship diagram (ERD). Therefore, when system designers change the base object model, subsequent changes in the database model can also be required.

2.2.1.9 Methodology Summary

From an initial requirements document, and with user collaboration, we defined the proper use cases and built a GUI prototype using VisualAge. The use cases and the GUI prototype together represent the requirements model for the problem to be solved. The requirements model also defines the system boundary and external interface.

From the requirements model and the problem statement, we build the first-cut object model. This object model gives an overview of the application classes that describe the problem domain, their relationships, and their base attributes.

We then built CRC cards to define class responsibilities and collaborators. From class responsibilities, we determined the services and attributes in the object model. From collaboration patterns, we defined and refined relationships, in the object model.

Referencing the object model, from use cases and required services, we built event trace diagrams and state diagrams showing system dynamic behavior.

Table 1 summarizes the analysis tasks in our approach.

Table 1. Analysis Phase

Input	**Techniques and Tools**	**Deliverables**
Application requirements specifications -Input/output -Potential users -User input	Use case gathering	Use cases (first cut) Actor(s)
Use cases (first cut) -User input	Analysis prototyping with VisualAge -For requirements gathering -For proof of concepts	Use cases (refined), GUI prototype
Application requirements - Problem statement - Domain description	Syntactic analysis -Underlining nouns -Underlining verbs -...	List of candidates (stati -Classes and attribute -Relations
Use cases (refined)	Syntactic analysis -Underlining nouns -Underlining verbs -...	List of candidates (dynam -Classes and attribute -Relations
List of candidate (static + dynamic) classes and relationships	Pruning	List of good classes and relationsh
List of good classes and relationships	Preparing precise description of each object class	Data dictionary for the c
List of good - Classes - Relations	Object modeling	Object model (first cut)
Object model (first cut)	CRC - Class roles and responsibilities - Collaborations	Object model (1st iterati - Interfaces of classes (responsibilities) - Additional attributes
Object model (1st iteration) Use cases (refined)	Dynamic modeling	Event trace diagrams State diagrams - Pre- and post-conditio of event-class intera
Event traces Pre- and post-conditions Class responsibilities	Functional modeling	Algorithms of methods
Object model Dynamic model Algorithms Data dictionary	Iterate ...	Object model (refined) Dynamic model (refine Algorithms (refined) Data dictionary (refi

2.2.1.10 Integrity Rules

The rules listed below should be used to ensure that models produced during analysis are consistent. We applied these rules as review guidelines during the assessment phase in each iteration.

Object Model (OM) versus CRC Cards

- OM class maps to a CRC card
- OM role name maps to CRC collaborator
- OM method name maps to CRC responsibility

Event Trace versus CRC Cards

- OM class maps to a CRC card
- OM message from object A to B <-> CRC for A declares B as collaborator
- OM message from object A to B <-> CRC for B declares that B has the responsibility to service message

Object Model versus VisualAge Design Prototype

- OM class maps to nonvisual VisualAge component
- OM association role name maps to VisualAge component attribute
- OM attribute name maps to VisualAge component attribute
- OM method name maps to VisualAge action.

2.2.2 Sample Application: Analysis Work Products

This section describes the work products of the analysis stage of the sample application:

1. Use Cases
2. Graphical User Interface prototype
3. CRC (Classes/Responsibilities/Collaborators)
4. Data Dictionary
5. Object Model
6. Dynamic Model
7. Sample Application Prototype

Subtopics

2.2.2.1 Use Cases
This section describes the use cases that were defined for the sample application:

Customer Order Management--In Stock

1. The cashier creates new order

2. The cashier populates order (currency type, amount, amount type (cash/check), type of payment, customer information).

    - ***Out of the use case*** (6)
    - The system determines currency type based on country.
    - The order can handle multiple currencies and checks.
    - The system can determine country restrictions.

3. The system tells cashier re: stock availability - in stock case.

4. The system determines payment and exchange rate.

5. The system prints tab.

6. The cashier notifies system; customer accepts (signed) order.

7. The cashier handles customer payment - points to another use case:

    - ***Out of the use case***
    - The system reduces stock level of currency type and amount of order.
    - The system generates accounting entry.

8. The system notifies cashier order ok.

9. The cashier completes order.

Customer Order Management--Out of Stock

1. The cashier creates new order.

2. The cashier populates order (currency type, amount, amount type (cash/chk), type of payment, customer information).

    - ***Out of the use case***
    - The system determines currency type based on country.
    - The order can handle multiple currencies and checks.
    - The system can determine country restrictions.

3. The system tells cashier re: stock availability - out of stock.

4. The cashier places out of stock order

    - ***Out of the use case***
    - The system determines payment and exchange rate.

5. The system prints tab.

6. The cashier notifies system; customer accepts (signed) order.

7. The cashier takes deposit if noncustomer:

    - ***Out of the use case***
    - The system forwards the order to the bank.
    - The system passes accounting entries to the bank.
    - The bank handles order--points to another use case.

8. The system notifies cashier order pending.

9. The cashier inquires pending orders when customer comes in.

10. The cashier handles customer payment--points to another use case.

11. The cashier completes order after customer receives cash or check.

Customer Sells to Branch Cashier

1. The cashier creates "sell" order.

2. The cashier populates order (currrency type, amount, denomination, customer information).

3. The system tells cashier qualitative information about currency; country, denomination, description of currency, common forgery errors.

    - ***Out of the use case***

    - The system determines exchange rate, payment.

4. The systems prints tab for customer and bank receipt.

5. The cashier examines currency.

6. The cashier accepts order:

    - ***Out of the use case***

    - The system adds stock to stock totals.

    - The system passes accounting entries to the branch or bank.

Branch Management--Stock Replenishment: The actor for this use case is the branch manager (or this could be a time-initiated batch function).

1. The branch manager gets from the system all requests of Cashier Stocks (can be replenishment orders or requests to send excess stock).

2. The branch manager creates a consolidated order to the bank for replenishment or return of stock.

3. The branch manager forwards the order to the bank.

Cashier Management--Reconcilation Required

1. The cashier checks stock in drawer:

    - ***Out of the use case***

    - The system obtains exchange rate for each currency.

2. The system displays each currency and check totals and local currency equivalent.

3. The system displays each currency or check denomination totals and local currency equivalent.

4. The cashier verifies stock in drawer with stock in system--stocks balanced.

5. The cashier raises compensating accounting entries for loss and gain:

    - ***Out of the use case***

    - The system amends stock accordingly

    - The system archives reconcilation records

6. The system verifies and informs cashier whether minimum stock quantity reached.

7. The cashier orders replenishment stock.

8. The system verifies and informs cashier whether the maximum stock quantity has been exceeded.

9. The cashier sends excess stock to center.

10. The system displays total local currency equivalent in stock.

Cashier Management--No Reconcilation Required

1. The cashier checks stock in drawer:

   - ***Out of the use case***

   - The system obtains exchange rate for each currency.

2. The system displays each currency and check totals and local currency equivalent.

3. The system displays each currency or check denomination totals and local currency equivalent.

4. The cashier verifies stock in drawer with stock in system--stocks balanced.

5. The system verifies and informs cashier whether minimum stock quantity reached.

6. The cashier orders replenishment stock.

7. The system verifies and informs cashier whether maximum stock quantity exceeded.

8. The cashier sends excess stock to center.

9. The system displays total local currency equivalent in stock.

Use Case Model: A sample use case model is shown in Figure 20.

PICTURE 20

Figure 20. Cashier of Branch Management System

(6) The use case describes the interaction between the user and the system. What happens inside the system is either described in the problem statement or has to be elicited from user or expert knowledge.

2.2.2.2 CRC

The index cards represented below show the results of our CRC exercise.

Class: Order

Responsibilities	**Collaborator**
Identifies customer, cashier, branch	Cashier, Customer, Branch
Knows and can modify currency, check and amount requested	
Knows serial numbers of issued T-Check	
Knows status (new, pending, completed, cancelled)	
Calculates payment-amount	Currency(x rate), Denomination(amount), Branch(commission)
Checks country restrictions	Country

Class: Country

Responsibilities	**Collaborator**
Knows restrictions	
Knows currency type	

Class: Currency

Responsibilities	**Collaborator**
Contains set of denomination type	DenominationType
Knows exchange rate	

Class: Bill-Type

Responsibilities	**Collaborator**
Knows its value	
Knows currency type	Currency

Class: T-Check-Type

Responsibilities	**Collaborator**
Knows its value	
Knows currency type	Currency

Class: Denomination

Responsibilities	**Collaborator**
Knows description	Currency
Knows currency	Currency
Knows value	

Class: Cashier

Responsibilities	**Collaborator**
Identifies the branch worked in	Branch
Knows own name	
Creates customer order	

Class: Branch

Responsibilities	**Collaborator**
Knows the Cashiers working in the Branch	
Holds branch name	
Replenishes branch stock	BankStock

Class: CashierDrawer

Responsibilities	**Collaborator**
Knows availability of cash and t-checks	
Knows serial number of available t-checks	
Adds additional currency (checks/cashs)	
Removes currency (checks/cashs)	
Can create a list of depleted stock items (qty < min)	
Can create a list of excess stock items (qty > max)	
Knows minimum/maximum quantity per currency	

Class: BranchStock

Responsibilities	**Collaborator**
Knows availability of cash/check in the branch	CashierDrawer
Can create replenishment orders if branch is low or out of stock	CashierDrawer, BranchRese
Can create order to send excess branch stock to the central bank	CashierDrawer, BranchRese
Accept and distribute additional currency (checks/cash)	BranchReserve, CashierDra

Class: BranchReserve

Responsibilities	**Collaborator**
Knows availability of cash and t-checks	

Knows serial number of available t-checks
Adds additional currency (checks/cashs)
Removes currency (checks/cashs)
Can create a list of depleted stock items (qty < min)
Can create a list of excess stock items (qty > max)
Knows minimum/maximum quantity per currency
Reports availability of stock items from list of requested stock items, for example, request: 5000 pesos, answer canSupply: 2000 pesos

Class: Bank

Responsibilities	**Collaborator**
Holds bank stock	BankStock
Holds bank name	
Knows all branches	Branch
Replenishes bank stock	

Class: Account

Responsibilities	**Collaborator**
Knows owning customer	Customer
Holds balance	
Credit/debit for deposit and payment	Payment, Deposit

Class: Customer

Responsibilities	**Collaborator**
Knows name	
Places order	
Deposit and payment	Deposit, Payment

Class: StockItem

Responsibilities	**Collaborator**
Holds stocked item and quantity	

Class: BankReserve

Responsibilities	**Collaborator**
Holds reserve stock for the bank	

Class: BankStock

Responsibilities	**Collaborator**
Knows the available stock of the bank	Bank

Class: Payment

Responsibilities	**Collaborator**
Accepts deposits	Deposit
Pays customer purchased order	CustomerOrder

Class: CashPayment

Responsibilities	**Collaborator**
Holds cash deposit for non-account customer	

Class: AccountPayment

Responsibilities	**Collaborator**
Credit account with deposit	Account
Debit account with payment	Account

Class: CustomerOrder

Responsibilities	**Collaborator**
Holds date created	
Holds total ordered amount	

Class: OrderItem

Responsibilities	**Collaborator**
Holds order items and quantity	

2.2.2.3 Data Dictionary

We came up with the following descriptions for the object classes we identified in the context of the FCE application:

Account - an account in a bank against which credit or debit transactions can be applied. Customer who owns an account in the bank can deposit into and make payment from the account to pay for the foreign currency purchase orders.

Bank - a financial institution that holds accounts for customers and holds the stock of foreign currencies for the purpose of providing foreign currency exchange service to customers.

BankReserve - an inventory of foreign currencies in the bank saved for future use in the foreign currency exchange service.

BankStock - an inventory of foreign currencies in the bank, including both the portion in circulation and the portion saved for future use in the foreign currency exchange service.

BillType - a unique type of the set of bills representing a currency, which has a specific value.

Branch - an establishment of a bank that provides foreign currency exchange service to the bank customers.

BranchReserve - an inventory of foreign currencies in the branch saved for future use in the foreign currency exchange service.

BranchStock - an inventory of foreign currencies in the branch, including both the portion in circulation and the portion saved for future use in the foreign currency exchange service.

Cashier - an employee of a bank working in a branch, who is authorized to create customer orders for purchasing foreign currencies and accept customer deposit and payment for the order.

CashierDrawer - a temporary storage containing foreign currencies used by a cashier to conduct foreign currency exchange business.

Country - a state or nation which has its own currency. A country may have restriction or policies in the amount of currency a traveler can get.

Currency - money in circulation as a medium of exchange, acceptance and general use. A country has its own currency which consists of a set of denominations. The medium of a currency includes bills and traveler's checks (in the context of this application).

Customer - a client who has an account with the bank to purchase foreign currencies.

DenominationType - a unique type of the set of units which identifies a specific type and value in the currency system.

Order - a request for purchasing or supplying an amount of foreign currencies with itemized specifications.

OrderItem - an item in the order which makes up an order.

Payment - a transaction to pay for a foreign currency order.

TCheckType - abbreviation for Traveler's Check Type - a unique type of traveler's checks in a currency which represents a specific value.

Stock - the supply of foreign currencies kept on hand by a financial institution (bank) or an establishment (branch) to conduct foreign currency exchange business.

StockItem - an item in the stock which represents a specific type and amount of the supply in foreign currencies.

2.2.2.4 Object Model

The initial version of our object model is shown in Figure 21.

```
+------------------------------------------------------------------------+
¦                                                                        ¦
¦                                                                        ¦
¦                                                                        ¦
¦ PICTURE 21                                                             ¦
¦                                                                        ¦
¦                                                                        ¦
+------------------------------------------------------------------------+
```
Figure 21. Initial Object Model for the Branch Management System

2.2.2.5 Dynamic Model

The first cut of the state diagram for the order class is shown in Figure 22.

```
+------------------------------------------------------------------------+
¦                                                                        ¦
¦                                                                        ¦
¦                                                                        ¦
¦                                                                        ¦
¦ PICTURE 22                                                             ¦
¦                                                                        ¦
¦                                                                        ¦
¦                                                                        ¦
+------------------------------------------------------------------------+
```
Figure 22. First-Cut State Diagram for the Order Class

The sample event trace and state diagrams that constitute the dynamic model are shown in Figures 23 through 25.

```
+------------------------------------------------------------------------+
¦                                                                        ¦
¦                                                                        ¦
¦                                                                        ¦
¦                                                                        ¦
¦ PICTURE 23                                                             ¦
¦                                                                        ¦
¦                                                                        ¦
¦                                                                        ¦
+------------------------------------------------------------------------+
```
Figure 23. Sample Dynamic Model

```
+------------------------------------------------------------------------+
¦                                                                        ¦
¦                                                                        ¦
¦                                                                        ¦
¦                                                                        ¦
¦ PICTURE 24                                                             ¦
¦                                                                        ¦
¦                                                                        ¦
¦                                                                        ¦
+------------------------------------------------------------------------+
```
Figure 24. Sample Integrated Dynamic Model

```
+------------------------------------------------------------------------+
¦                                                                        ¦
¦                                                                        ¦
¦                                                                        ¦
¦                                                                        ¦
¦ PICTURE 25                                                             ¦
¦                                                                        ¦
¦                                                                        ¦
¦                                                                        ¦
+------------------------------------------------------------------------+
```
Figure 25. Event Trace Diagram for Stock Replenishment

2.3 Chapter 7. Designing and Constructing the Solution

After we modeled the problem domain for the FCE application, we proceeded with designing and building a solution for it.

As discussed in "Object-Oriented Design Considerations" in topic 2.1.2, object-oriented design encompasses both *system design* and *object design* activities. System design in a Client/Server environment is a vast and complex topic. In this chapter, we highlight only the main Client/Server system design considerations for the FCE application. Our main focus is on the object design aspects with VisualAge. Our intent is to suggest design and development guidelines in using VisualAge with object-oriented design approaches.

Subtopics

2.3.1 Object-Oriented Design: An Integrated Approach

Presented here is an approach we developed that integrates VisualAge into the object-oriented application design process. It was used to develop the FCE application. This design approach is summarized as follows: (7)

System design

- Decompose the analysis stage object model into subsystems, which then provides the strategy for partitioning the application into subapplications, as part of our high level application architecture design.

- Choose a Client/Server platform and enabling technology, as the basis for deriving a high-level system architecture.

Object design and prototyping using VisualAge

- Map the semantic classes identified in the analysis object model to solution domain classes (for example, application classes, service classes), which become the required VisualAge nonvisual classes. (8)

- Design the details of the VisualAge nonvisual classes, defining and refining:

  - The nonvisual classes' public interface (attributes, actions, events) and methods
  - Nonvisual classes' instance methods and variables
  - Derived attribute policies.

- Design the VisualAge visual classes for the GUI of the application, designing and refining:

  - The elementary visual class for each application class
  - Additional composite visual classes as required
  - Input data validation
  - Deferred updates.

- Iterate on the design prototype.

Persistent data access and update design: When the prototype shows an acceptable degree of functionality, evolve it to a working solution, adding the design for:

- Unit of work

- Persistent object, data storage and access:

  - Model and design server databases
  - Define the interactions between database objects and model objects

- Distributed object support:

  - Define the distribution of objects in users' images
  - Define access policies for retrieve/update shared data

VisualAge is a powerful tool that allows a prompt implementation of design decisions, so we do not make a clear cut separation between design and implementation issues. Instead, we discuss our design ideas and show how they can be translated into a VisualAge implementation.

In the sections that follow, we discuss in more detail our *system* and *object design* and *persistent data access design* processes.

(7) Object-oriented design is an iterative process. These design steps are presented here in order, but they are carried out iteratively.

(8) Application classes are the result of mapping the semantic classes (these are the problem domain classes) identified in the object-oriented analysis to the solution domain. They include both the base classes, which have meaning by themselves, and relationship classes, which represent the meaning of the relationships between two base classes.

Service classes are domain independent classes that are used for system related functions, such as database access, communication interfaces, etc.

2.3.2 System Design

System design is the design of a high-level architecture for the proposed solution. This includes a definition of the major system building blocks and their high-level connectivity and an application architecture that organizes the solution in subsystems for the allocation of these subsystems to the system building blocks. The building blocks reflect system functions, as opposed to hardware or software products.

As part of system design, we need to make the initial design decisions for the placement of data and processing and select the system platforms to implement the solution. Taking into consideration the available information technology (IT) current environment, such as existing legacy systems, design decisions are made to select the enabling technology and components for implementing the major system build blocks.

The main activities for the system design stage are shown in Table 2.

Table 2. Design Phase: System Design Overview

Input	**Process, Techniques, Tools**	**Deliverables**
Object Model, Dynamic Model Event Trace Diagrams	System partitioning	High-level system archite Subsystems Subsystems interactic
Object Model Subsystems	Mapping	VisualAge application subapplications
Subsystems Common data access requirements Performance considerations Replication Legacy system ?	End-to-end system design	System platforms selectic and design decisions f Object placement (OO Data/function placeme server)

Subtopics

2.3.2.1 Partitioning the Object Model into Subsystems

The term subsystem is used in OMT and RDD to refer to a grouping of tightly coupled classes.

There are many reasons to split the object model into subsystems, the main ones are to:

- Manage the complexity of the design effort
- Split the design effort in more than one team
- Isolate specific design decisions that the designer believes can be later changed
- Distinguish different levels of abstraction in the services provided

The first goal can be successfully met if a subsystem is not just a bunch of classes but provides some useful abstraction to better understand the structure of the application.

The second goal can be successfully met if the dependencies between design teams are loose (they do not depend too much on each other's design decisions). This can be achieved only if the message flow between subsystems is much simpler than the message flow within subsystems.

A guideline to follow in decomposing the application's object model into subsystems is: *decompose in such a way that the message flow between subsystems is minimized*.

A sample metric for good subsystems selection

msgSent	Set of messages with senders in a class of a subSystem
msgGoingOut	subset of msgSent having implementers in another subSystem
msgStayIn	subset of msgSent having implementers in the same subSystem

The subSystem is good if msgStayIn >> msgGoingOut

A heuristic approach to find a good set of initial candidate subsystems is a *responsibility driven approach*: decompose in such a way that a well-defined subset of the application responsibilities can be assigned to each subsystem.

2.3.2.2 Mapping Subsystems to VisualAge Subapplications

The term subapplication is used in the VisualAge team programming environment as divisions for an application. How do we determine our VisualAge subapplications? Although the meaning of VisualAge subapplication does not have the same semantics as subsystem, we choose the subapplications to implement in VisualAge based on the subsystems derived from the object model.

2.3.2.3 Selecting the Implementing Platform

Selecting the system platforms or infrastructure is required to make the application design workable and manageable. In our case, the implementing platform is assumed to be an OS/2 DB/2 2 LAN environment. The enabling technology and component selections for our system building blocks are predetermined given the available supporting platform of VisualAge at the time of our project.

2.3.2.4 Data and Function Placement

The initial decisions for the placement of data and processing are made during system design. These decisions will be reassessed during the object design stage.

2.3.3 Object Design

The object design phase includes a refinement and a fleshing out of the object details. The main deliverables are described in Table 3.

Table 3. Design Phase: Object Design Overview

Input	**Process, Techniques, Tools**	**Deliverables**
Object Model (classes)	Design classes - Map semantic classes to VisualAge nonvisual classes - Add interface and service classes	VisualAge nonvisual class
Object Model (services and attributes)	Design public interface for VisualAge nonvisual classes	VisualAge public interfac (attributes, events, a
Object Model (associations), VisualAge nonvisual classes	Design association implementation	- Refine VisualAge publ interfaces (more attr - Smalltalk implementat collection classes
Base classes, GUI Prototype	Design elementary GUI components	VisualAge elementary visu
Base classes, VisualAge elementary visual classes	Design composite GUI components	VisualAge composite visua
Use cases, GUI prototype, VisualAge elementary and composite visual classes	Build application GUI	VisualAge end user visual

Subtopics

2.3.3.1 Design the Solution Domain Classes

The set of object classes that make up a running application is usually much larger than the set of classes identified during the analysis phase. The initial set of semantic application classes identified in the analysis object model represents only the "core" business behavior of the application. Other solution domain classes must be designed to provide concrete functionality of the application. Interface classes that represent the user interface and "service" classes that provide services functions such as input data validation and database access are some examples of additional classes required for the implementation of the application.

The design of solution domain classes is, as everything else in object-oriented development, iterative. However, we suggest the following design steps:

1. Map the semantic application classes, identified in the object model from analysis, to VisualAge nonvisual classes--this is almost straightforward.
2. Add interface and "service" classes to provide user interface and additional functionality as required--as explained below.

For each application class in the analysis model, an elementary user interface is added to verify its functionality. The elementary user interface classes can then be constructed with VisualAge as potential reusable view classes--they are reused to form the more sophisticated composite views.

The result of this design activity will lead to additional solution domain classes, including:

- Interface classes, to provide user access and interface (9)

- Service classes, to provide database access, system services functions, and other operations

In general, we follow the principle to evenly distribute the responsibilities among objects (see :bibref refid=wir90.). And we try to avoid introducing specialized control objects to keep the solution robust and flexible.

Figure 26 shows a mapping from the object model to VisualAge and Smalltalk constructs during the design. We explain the design process in more detail in the sections that follow.

```
+--------------------------------------------------------------------------+
¦                                                                          ¦
¦                                                                          ¦
¦                                                                          ¦
¦                                                                          ¦
¦ PICTURE 26                                                               ¦
¦                                                                          ¦
¦                                                                          ¦
¦                                                                          ¦
+--------------------------------------------------------------------------+
```

Figure 26. Mapping from Object Model to VisualAge and Smalltalk Constructs

(9) The interface classes will evolve to become the VisualAge visual parts.

2.3.3.2 Detail Design for the VisualAge Nonvisual Classes

Once the required solution domain classes are determined, we are ready to flesh out the details for each of them.

<u>VisualAge Design Considerations</u>: Let us first examine the issues that must be addressed during VisualAge application design independently from the target environment (Client/Server or single user):

- Choosing the right data structures to support object relationships: Smalltalk provides a powerful bag of predefined data structures that can be used to support object relationships. The choice is dictated by semantic considerations (can I keep duplicated information? Does my data imply any order?) as well practical considerations (how much data will be managed?).

- Designing derived attribute policies; it is useful to separate the attributes of an object into two categories: *primitive attributes* are attributes that cannot be derived from other attributes of an object (for example, the items ordered are a primitive attribute of an order) and *derived attributes* are attributes that can be derived from other attributes (for example, the total value of an order).

  Proper policy should be designed to guarantee that derived attribute values are always congruent with the values of the primitive attributes and to assign responsibility for the calculation of derived attributes.

- Logical data integrity policy design: a great deal of the application programmer's work is devoted to building controls on input data from the user. Who (which object) should be responsible for verifying input data? Various alternatives can be devised:

  - Target objects expect that data is correct when passed to them and the burden of verifying it is on the sender object
  - Target objects verify data when they are asked to make any modification. to their state.

During the analysis phase, we defined for each class in the object model the attributes (the knowledge that the object is responsible for maintaining) and the services the object must provide to other classes. The object model also shows the associations between classes (the knowledge that classes of objects have of each other).

From this information, a design for the VisualAge nonvisual classes can proceed with the following two steps: First, *define the public interface for each nonvisual class*; and then *design the Smalltalk methods and instance variables* needed to support the public interface.

<u>Define the Public Interface</u>: The VisualAge public interface defines the external characteristics of a VisualAge part. In other words, the public interface of a VisualAge part defines how other components can interact with it.

The VisualAge public interface is made up of:

- Attributes, which define properties of a component that are accessible by other components through its get selector method.

- Actions, which define services implemented by a component that can be requested by other components.

- Events, which define services implemented by a component that can signal the occurrence of events to other components.

In the following discussion we use the terms "OM attributes," "OM services," and "OM associations" to refer to the corresponding elements defined in the object model.

The public interface for each nonvisual class is designed according to the following rules of thumb:

- OM attributes become VisualAge attributes, distinguish between:

  - Primary attributes - these attributes can be read and set.
  - Derived attributes - these attributes can be read but not set, so they should not have set selectors when implementing them in VisualAge.

- OM services become VisualAge actions, except for:

  - OM services that just provide requested information; these

services can better be implemented as VisualAge attributes.

An example of this kind of service is the provision of list of depleted items in a stock in our FCE application.

- For each service provided, identify:

  - The preconditions that must be verified for a service to be executed--define a VisualAge event to signal that the precondition is not met.
  - The postconditions that must be verified when a service is successfully executed--define a VisualAge event to signal that the postcondition is met.

- OM associations map to VisualAge attributes.

Mapping Associations to VisualAge Attributes: OM associations indicate the relationships between object classes. During object modeling we define the roles that an object plays in an association. For example, two persons can be related by a "marriage" association. One of them plays the "husband" role, and the other the "wife" role.

We map the roles played in an association to VisualAge attributes as hereby described, with an example to illustrate the process. Look at a fragment of the object model shown in Figure 27; suppose we are designing the external interface of Currency.

```
+--------------------------------------------------------------------------+
¦                                                                          ¦
¦                                                                          ¦
¦                                                                          ¦
¦                                                                          ¦
¦ PICTURE 27                                                               ¦
¦                                                                          ¦
¦                                                                          ¦
¦                                                                          ¦
+--------------------------------------------------------------------------+
```
Figure 27. A Fragment of the Object Model for the Currency Class

- There is an association between Currency and Bank (this association represents the fact that the Bank trades some currencies): Bank plays no role for Currency (that is, Currency is not going to collaborate with the Bank for any of its responsibilities).

  So we will not add any external attribute representing the Bank in the public interface of Currency.

- There is an association between Currency and Country. Country plays a role for Currency (in fact Currency is responsible for identifying the corresponding Country when required); and Currency plays a role for Country (its currency).

  So we add an external attribute representing the Country in the public interface of Currency; the new attribute is called country and is of class Country. And we add an external attribute representing the Currency in the public interface of Country; this attribute is named currency in the Currency class.

- There is an association between Currency and DenominationType (this association represents the fact that a Currency is actually composed of different types of bills and travelers' checks): Currency needs to know about DenominationType, in fact it is its responsibility to provide the list of available denominations when required.

  So we add an external attribute called denomination to Currency. Denomination will be of class Set because repetition is not allowed.

The process described in this example can be summarized in the following rules:

For a class (let's call it OurClass) in the object model, look for the classes to which this class is connected through some association (for example, AssociatedClass). Distinguish between:

- OurClass needs to know about just one instance of the AssociatedClass (the AssociatedClass plays some role for OurClass and AssociatedClass participates in the association with multiplicity 0 or 1). Add an external attribute in the interface of OurClass to represent the AssociatedClass. The name of the attribute should reflect the role that AssociatedClass plays for OurClass, and the class of the attribute will be the same of the AssociatedClass.

- OurClass needs to know about more instances of the AssociatedClass

(the AssociatedClass plays some role for OurClass, and AssociatedClass participates in the association with multiplicity 0 or 1 or more). Add an external attribute in the interface of OurClass to represent the collection of instances of AssociatedClass that OurClass needs to know. The name of the attribute should reflect the role that this collection of AssociatedClass instances plays for OurClass, and the class of the attribute will be a properly chosen subclass of Collection.

- OurClass does not need to know about AssociatedClass (the AssociatedClass does not play any role for OurClass). Do not add any attribute representing the AssociatedClass in the OurClass interface. It would be unnecessary information, which will have to be managed without adding anything to an application's behavior.

Choosing the Right Class to Represent Collection: How shall we choose a proper subclass of Collection to represent associations with multiplicity > 1?

In the example we choose Set as the class for denominationType; but are there other design alternatives?

Collection is perhaps one of the most powerful features of the Smalltalk environment. We sketch here briefly the Collection class and its subclasses:

The Collection class and subclasses

Collection	is an abstract class (that is, defines common characteristics of its subclasses but is never directly used)
Bag	keeps a collection of elements no matter if repeated or not
Set	keeps a collection of elements that cannot be repeated
Dictionary	keeps a collection of elements that are identified by a key
OrderedCollection	keeps a collection of elements that are identified by an index
SortedCollection	keeps a collection of elements that need to be kept in a sorted order
Array	keeps a finite size collection of elements that are identified by an index.

Choose Dictionary when fast access for a key value is required. Associative objects typically have a key. For example, the StockItems of a Stock are keyed on the Item.

Choose Array when the association has a finite multiplicity. For example, a car can have at most five passengers, so a "passengers" attribute for a Car class will be of type array.

Choose Set in all other cases.

Other members of the Collection family are used to build derived attributes: OrderedCollection is used to build lists of objects to be shown to the user, and SortedCollection is used to build a sorted list.

Optimizing the Use of Collection: The Smalltalk Collection class and its subclasses are very powerful but must be used with care: Smalltalk objects reside in virtual memory, so Collection operations expect to have the full collection available, but often collections are just too big to be loaded in memory.

For example, at first glance it seems that we could implement the association between Bank and Customer by adding an attribute *customers* of class Set in the Bank. A practical consideration that we must be aware of is that we will never be able to have the full set of customers loaded in memory because it is too big. Therefore, we will not be able to provide the user with a customer list that is built on an OrderedCollection derived from a Set class containing all customers. A good technique in that case is to use a qualified association, where the qualifier is, for instance, the customer number.

Design Smalltalk Methods and Instance Variables: We now discuss the design of the Smalltalk methods and instance variables to support the public interface.

Designing Derived Data Policy: Derived data is often required to build a usable application. For example, the user interface needs a list of the ordered items to allow the cashier to modify the order; the branch stock management needs a list of the depleted items from the cashier drawers to be able to build a consolidated branch replenishment order.

The problem can be seen in this way: there is a client object that needs to derive data from some primitive data known by a server object.

First decision: *who derives data*? Shall the server directly provide the derived data or just the base data leaving to the client the job of deriving what it needs?

Our approach is, in general, to *assign the responsibility to derive data to the object that owns primitive data*, that is, the server object. In this way we can obtain more information hiding (allowing for easier maintainability) and "richer" objects. A pragmatic decision should be taken according to the following considerations:

- Is the derived data needed only by this client?

- Does the derived data depend on any server design decision that is likely to change in the future?

VisualAge nonvisual class will have an external attribute declared for any piece of derived data required.

Second decision: *saving derived data*. Different approaches can be followed:

- Each time the client requests derived data, the server obtains it from base data (for example, each time a client requests a total on an order, the order loops through all order items and recalculates it). This means that the derived data is up to date every time it is requested.

- Derived data is obtained and saved by the server object. Derived data can be obtained by the server object:

  - Each time the base data (that is, the data used to calculate the derived data) changes. This means that every time a client requests derived data, this data is up to date, but the server object must keep track of which derived attributes need to be updated for any base attribute.

  - When requested, that is, the server provides "refresh" services only if a client requests the value of a derived attribute whose base components have been updated. This policy can be appropriate if deriving data has a high overhead.

The choice between the alternatives is dictated by storage space and response time constraints, and it usually implies a trade-off between both constraints.

Third decision: *push or pull*. VisualAge supports events, that is, a server object can signal to interested parties that a derived attribute is changed. This forces any attribute connected through an attribute to attribute connection to be refreshed (push). Alternatively, the server object can just provide data when required, and it is up to the client to require (pull) fresh data when needed.

An example can help in understanding how the various alternatives can be implemented with VisualAge, as shown in the following:

Sample implementation for derived data policies

Let's build a simple component, an Adder that has two primitive attributes: firstNumber and secondNumber; and a derived attribute: result. Result is derived with the formula: result := firstNumber + secondNumber.

Refer to Figure 28. The following sample implementations are proposed:

- AdderNoSavePull:
  - Calculates the result only when required
  - If some primitive attribute changes, no action is performed; it is up to the client to request a fresh copy of result.

  Public interface attributes: firstNumber, secondNumber. Public interface actions: result. Smalltalk code requested: code the result method to return the sum.
- AdderNoSavePush:
  - Calculates result when required
  - If some primitive attribute changes, an event is raised to signal to the clients the change

Public interface attributes: firstNumber, secondNumber, result.
Smalltalk code requested: in the set selectors of firstNumber and secondNumber add "self signalEvent: #result" to signal that result is changed.

- AdderSaveWhenNeededPush:
  - Calculates result when it is necessary and keeps it up to date
  - An event is raised to signal the change.

  Public interface attributes: firstNumber, secondNumber, result.
  Smalltalk code requested:
  - Add a new method "recalculateResult" with code: "self result: firstNumber + secondNumber"
  - In the set selectors of firstNumber and secondNumber add "self recalculateResult".
- AdderSaveWhenRequestedPush:
  - Calculates result only when requested
  - An event is raised to signal the change

    Public interface attributes: firstNumber, secondNumber, result. Public interface actions: recalculateResult Smalltalk code requested: add a new method "recalculateResult" with code: "self result: firstNumber + secondNumber"

Notice that:

- AdderNoSavePull requires no programming effort but somehow complicates the view's job. In this case we cannot define "result" as public attribute (?????? bug ???????) and we are forced to define a "result" action that returns the desired value; in our opinion this is not a very clean design.
- AdderNoSavePush produces a very clean view-model interface but requires a modification in the set selector of all primitive attributes.
- AdderSaveWhenNeededPush has the same characteristics of AdderNoSavePush but may be faster if the derived attributes are requested many times.
- AdderSaveWhenRequesedPush has the same characteristics of AdderNoSavePull but is highly recommended if derived attributes calculation is expensive.

PICTURE 28

Figure 28. Different Policies to Derive Attributes

Input Data Validation: Any interactive system must implement a complete validation of data entered by the end user. Some design decisions have to be made on who verifies data and when.

Two alternatives can be devised:

- Server object expects that data is correct when passed to it and the burden of verifying the data is on the client. This approach would comply with structured programming principles (see :bibref refid=wir90.) but somehow violates the encapsulation principle, because knowledge of the inner workings of an object is put outside it.
- Server object verifies data when it is asked to make any modification to its state. This approach, while properly assigning the knowledge of which state is acceptable for an object to the object itself, triggers verification also when servers are modified in a "friend environment" and a lot of useless controls are produced.

  Moreover, most update messages cannot be verified one by one; the whole update operation must be verified. For example, if a client wants to change the minimum quantity allowed for an item in a stock from 10 to 100 and the maximum quantity from 30 to 300, we are not expected to raise an error when changing the minimum quantity saying "minimum quantity greater than maximum quantity."

We suggest a compromise: it is the responsibility of any object to know whether a required update is acceptable to it, but it is a client responsibility to trigger the verification logic. This responsibility assignment allows for different levels of verification enforcement:

- Server in a friend environment (the update operation is surely correct and no verification is required). The client just updates the server object.
- Server in a hostile environment (the update operation must be verified, for example, the client is a user interface). The client asks the server whether the update it wants to do is acceptable; the server verifies the update request and asks the client to apply the change if everything is correct.

Figure 29 shows how the Client/Server update protocol can be designed for a logical point of view.

VisualAge has a DeferredUpdate object that provides the required function to implement this design.

PICTURE 29

Figure 29. Data Validation

2.3.3.3 Design the GUI with VisualAge Visual Classes

VisualAge provides a very powerful GUI framework for developing user interfaces. Its architecture strongly supports the Model-View separation design approach. It encourages component reuse by allowing composite components to be constructed using existing base components as subcomponents or by subclassing from existing components. The composite component provides encapsulation to all subcomponents allowing user interfaces to be constructed with relatively simple connections.

In the analysis phase, we have defined the OO model, in which the nonvisual classes but none of the visual classes are defined. In this design phase we will have to define the visual classes.

We recommend a bottom-up approach. We start at the bottom of the class hierarchy and we build one or more elementary views for each base object (nonvisual classes created previously). By elementary view we mean that the view is made up of only the base nonvisual component and the primitive GUI components. Most of these elementary views can be created very easily by using Quick Form (a menu choice on the connection menu). Once we have the elementary views created we start building composite views. A composite view is a user interface component made up of one or more of the elementary views built in the last step and other visual or nonvisual components. These elementary and composite views are the building blocks for the final assembly of the final end-user interface solution defined in the analysis prototype. In VisualAge each of these views is represented by a class, and we called them visual classes. The analysis prototype is used as a blueprint (specification) for identifying some of these reusable visual components.

Subtopics

2.3.3.3.1 End-User Interface Development

The following approach to develop an *end-user interface* (EUI) can be used in the majority of the application scenarios as a technique to identify visual classes.

Each concrete object (nonabstract class) has a nonvisual class (the model) and:

- A visual class (the view). Typically, it has at least one view and some have alternative views. For example, Customer object has one primary view that displays all the attributes about the customer. It also has an alternative view that displays just the proper name of the customer. Name is a derived attribute; the logic for displaying the proper name remains with the Customer class (some countries display last name first) and is transparent to the user of the composite Customer Name view component. See Figure 30 and Figure 31 for an example.

- May have edit and/or create view to allow entering of new or revised data. See Figure 32 and Figure 33 for an example.

- May also have a list or summary view displaying a list of existing objects. The list view typically displays only a subset of the attributes for each object. From the list view, you can operate on any of the selected objects. The typical operations are:

  - Open - to view all attributes of the object and, if required, update the object

  - Delete - to remove the object from the list

  - Add - to add a new object to the list.

  You can also locate an existing object through a Find dialog. See Figure 34 and Figure 36 for an example.

An application's GUI is constructed using a combination of such visual components. See Figure 35 and Figure 36 for an example.

PICTURE 30

Figure 30. Sample Elementary View: for Primary Use

PICTURE 31

Figure 31. Sample Alternative View: for Secondary Use

PICTURE 32

Figure 32. Sample Composite View: Edit View

```
¦                                                                          ¦
¦ PICTURE 33                                                               ¦
¦                                                                          ¦
¦                                                                          ¦
+--------------------------------------------------------------------------+
```
Figure 33. Sample Composite View: Create View

```
+--------------------------------------------------------------------------+
¦                                                                          ¦
¦                                                                          ¦
¦                                                                          ¦
¦ PICTURE 34                                                               ¦
¦                                                                          ¦
¦                                                                          ¦
+--------------------------------------------------------------------------+
```
Figure 34. Sample Composite View: List View

```
+--------------------------------------------------------------------------+
¦                                                                          ¦
¦                                                                          ¦
¦                                                                          ¦
¦ PICTURE 35                                                               ¦
¦                                                                          ¦
¦                                                                          ¦
+--------------------------------------------------------------------------+
```
Figure 35. Sample Application: Creating a New Customer

```
+--------------------------------------------------------------------------+
¦                                                                          ¦
¦                                                                          ¦
¦                                                                          ¦
¦ PICTURE 36                                                               ¦
¦                                                                          ¦
¦                                                                          ¦
+--------------------------------------------------------------------------+
```
Figure 36. Sample Application: List of Customers

Some design decisions that we made in our solution for the FCE application are:

- Use the forms metaphor as the common user interface style.

- Use consistent selection styles across different subsystems to minimize relearning.

  - Use notebook control and notebook page to provide access to an object instead of the conventional pull down menu.

  - Use context menu for choice of operation on an object. Push also provided as an alternative.

- Adopt the Model-View approach as supported by VisualAge. Encapsulate the model in composite view components. All business logic and validation rules are implemented in the model (nonvisual class).

- Design for maximum reuse and simplicity of connection. If, for instance, there are more than 10 connections and/or 10 subcomponents on the composition editor (that is, we have a complex component structure), it is advantageous to create a composite component to encapsulate some of the components and connections. The goal is to provide a public interface which should be easy and natural to use with Visual Programming. As a rule of thumb, we try to target not more than two attributes, two events, and two actions (not including the superclass external interfaces such as openWidget). For example, the CustomerEditView would have only one attribute, namely 'aCustomer,' and one event, namely 'customerChangedRequested'. The actual editing of input data, updating of customer record database, and so forth are encapsulated inside the component and transparent to the user of the component. The event is a postcondition event to be

used to trigger other actions as a result of this event. Another example is the CashierStockListView. In this case, it has two attributes, one, aCashier, and the other, selectedStockItem.

The implementation strategy that we adopted in our solution is as follows:

1. For each nonvisual class (base object)

    - Create a visual class for displaying the object (primary display view) using Quick Form or manually lay out the user interface. Look for any additional visual classes (alternative display views) that are required to assemble the final user interface (UI). Add the nonvisual class as a variable subcomponent and add it as an attribute to the public interface of this visual class.

    - Create a visual class for editing existing objects for this base class (primary edit view). This visual class may use the primary display view as a subcomponent or implement it as a subclass of the primary display view. To provide editing capability, the deferred update component can be used as a subcomponent in this class. The apply will be triggered by a noError event resulting from a verify action. Input data should be validated, and any necessary external events should be defined to the system.

    - Create a visual class for creating a new object for this base class (primary create view). This may be identical to the primary edit view in some cases. Use the primary edit view as a subcomponent or implement it as a subclass. In addition, use the object factory or our object cloner component to provide object template capability. (We use our own object cloner class. The Object Factory class creates new objects; it does not copy objects' attributes at any sublevel. We created our own ObjectCloner class that provides a deepCopy capability.)

    - Create a visual class for displaying a list of objects. Define any necessary external events.

2. For each subsystem, create visual classes as defined in the analysis prototype and construct the view by using the various visual classes created in the first part as building blocks. These are the interfaces users will use in the application. Define any necessary external interfaces which can then be used for the construction of the main application window.

2.3.3.3.2 Visual Class - Reuse

The object-oriented environment provides two kinds of code reuse:

- By delegation--sending a message to another class or object to get serviced. This is the same as sharing coroutines or common functions in a procedural environment.

- By inheritance--using the class hierarchy and reusing the services provided by the superclasses.

In VisualAge, you can reuse a visual class by adding it as a subcomponent to the component. This is the same as reuse by delegation. When the subcomponent class is changed, the component will pick up all the changes including any visual programming changes.

You can also create a visual class by subclassing from another visual class. Because a VisualAge visual class is also a regular Smalltalk class, in principle it behaves like reuse by inheritance. However, some VisualAge visual elements (such as attribute settings) are instance data and are not inherited. In general, you can only inherit Smalltalk-specific code. Most of the visual elements are not inherited. When you subclass from a superclass, you will get a copy of the superclass VisualAge specific instance data on the composition editor. Therefore, any visual changes made on the superclass will not be reflected in the subclass.

As a rule of thumb for visual classes, we find that the VisualAge environment encourages reuse by delegation, that is, adding the visual class as subcomponent. Unless the visual class has a significant amount of Smalltalk code and relatively simple visual elements that are highly unlikely to be changed, reuse by subclassing makes sense. Our experience is that most of our visual classes are reused by adding them as subcomponents. Reuse by inheritance is more suitable for nonvisual classes.

2.3.4 Design for Persistent Data

One of the more intriguing areas of object-oriented design is the mapping of persistent object data identified in an object model into a relational database design. The issues involved with this mapping add new challenges to the realm of database administrators, data modelers, and traditional systems designers.

This section describes the process of mapping a subsystem of the object model for the FCE application into a relational database design. With the examples listed, one should be able to apply the basic steps presented and use them in successful object-to-relational applications.

Subtopics

2.3.4.1 Three Schema Architecture

Perhaps one of the more important developments in database design methodologies was the publication of the ANSI three schema architecture (see Figure 37). This architecture solidified the data design approaches among database practitioners.

The architectural view suggests that data design should comprise three layers: the external, the conceptual, and the internal schemas.

The external schema is an application view or application abstraction of the conceptual schema. The external schema aids designers by simplifying the sometimes complex conceptual schema. Each external schema supports a data design for an application.

The conceptual schema integrates the external schemas. Through this integration the data relationships and data policies can be effectively built. These policies and relationships reflect the enterprise use of data, not just a specific application's data requirements.

The internal schema is the physical implementation of the conceptual schema's data requirements. These are usually expressed in terms of database management system DDL.

Object modeling is useful for designing both the external and conceptual schema of an application system. In fact, object modeling is a form of entity modeling :bibref refid=rum91..

```
+------------------------------------------------------------------------+
¦                                                                        ¦
¦                                                                        ¦
¦                                                                        ¦
¦                                                                        ¦
¦ PICTURE 37                                                             ¦
¦                                                                        ¦
¦                                                                        ¦
¦                                                                        ¦
+------------------------------------------------------------------------+
```
Figure 37. External, Conceptual, and Internal Schema Representations

2.3.4.2 From Object Model to Relational Database

In the discussion that follows, we focus on transforming part of the object model into a viable entity-relationship diagram (ERD). If we can translate the base object model into an ERD, we can also translate the ERD into a model that can be implemented by multiple relational database management systems. With this in mind, this section focuses primarily on ER model building and secondarily on the specifics of physical implementation.

<u>ER Modeling: Starting Relational Database Design</u>: In object-oriented application development, one may begin database design after a "rough" completion of the OOA. From OOA we obtain our first understanding of the base objects in the system, the base attributes of those objects, and the services that the objects provide.

Let's now take an in-depth look at how to map the object model to a relational database design.

<u>Building the ER Model from an Object Model</u>: We begin with the OOA object model (see Figure 19 in topic 2.2.1.6). Notice that the object model is incomplete. There are attributes missing, there are several many-to-many relationships, there are generalizations defined. These are elements that are traditionally seen in high-level data analysis. Models like this are not ready for direct database implementation. Reviewing the model, we can see that the database designers have a lot of work to do. To map the object model to an ERD, the designers need to:

1. SELECT an initial susbsystem to design

2. IDENTIFY the base entities from the object model

3. EXAMINE the object model relationships

4. MAP the object model relations to entities in the ERD

5. ADD new entities, add new relations

6. EXAMINE ERD and object model relationship types

7. MAKE design decisions based on relationship types

8. TRANSLATE the ERD model to a database implementation model

9. REVIEW changes in object model.

<u>Selecting a Subsystem</u>: The first step in relational database design is to choose a group of relations that appear to be closely bound from the initial OOA model. We choose a subsystem for two reasons. The first is to get a small and controllable start on the design process. The second is to integrate database design with the output of OOA.

Choose an isolated subsystem, if possible, that would lean toward a simplified test case implementation. Usually, this simplifies the initial design effort. Also, a good understanding of the design basics can enhance discussions throughout later iterations as the object model becomes more complex.

In database design terminology, consider the selected group of relations or subsystem as the external schema for a single application.

Let's choose the lower half portion of Figure 19 in topic 2.2.1.6. The group is of objects that pertain to stock. Let's call this the stock management subsystem. We have the following objects in our first database design iteration:

- Stock
- Denomination
- Cashier Drawer
- Branch Reserve
- Branch Stock

<u>Identify Entities</u>: For each class in the selected subsystem of the object model, map that class to an entity in the ERD. See Figure 38. In the top part of the figure both *Stock* and *Denomination* map to an individual entity. The attributes for each class also map directly to the entity.

```
+------------------------------------------------------------------------+
¦                                                                        ¦
¦                                                                        ¦
```

```
¦                                                                        ¦
¦                                                                        ¦
¦ PICTURE 38                                                             ¦
¦                                                                        ¦
¦                                                                        ¦
¦                                                                        ¦
+------------------------------------------------------------------------+
```

Figure 38. Object Model to Entity-Relationship Model

Examine Object Model Relationships: Examine the relationships between the object model classes.

Let's review relationship types and relationship multiplicity. Objects in the object model are related to one another through one of following relationship types:

- Aggregation
- Generalization
- Association.

Aggregation implies a "part-of" or "consists-of" relationship between two or more objects. Our example is *branch stock*. It "consists-of" *cashier drawers* and the *branch reserve*. Reading the inverse of the relationship, the *cashier drawers* and the *branch reserve* are "part-of" *branch stock*.

Generalization implies an "is-a" relationship between the subclasses and the superclass. An example is *stock*. See Figure 39. In this relation *cashier drawer* "is-a" *stock* and *branch reserve* "is-a" *stock*.

Association implies that there is a relationship between objects. This relationship is defined by its multiplicity and its name. An example is between *stockitem* and *denimonation*. See the middle portion of Figure 39. *Stockitem* "has-a" *denomination*. This association is classified as a binary one-to-many relationship. By a binary relationship we mean that only two entities are involved in the relation. One-to-many describes the multiplicity of the relation. For each *Denomination* there may be many instances of *Stockitem*.

There are several associations where the multiplicity is not one-to-many. The multiplicity of associations falls into the following categories:

- 0 --> 1
- 1 --> 1
- 0 --> many
- 1 --> many
- many --> many.

The inverse of these relationships is also considered valid.

The database design rules will differ significantly based on the multiplicity of the association. This is key to understanding and properly transferring the object model into an ER model.

In summary, by examining the relationships, we mean understanding the relationship type and, if pertinent, the multiplicity. This leads us into the next topic, mapping these relationships.

Map Object Relationships to Entity Relationships: The focus of the relationahip mapping is to accuratley capture the object model relationships and reflect them in the ERD.

Let's look at an example to illustrate this point. See the *stock* and *denomination* relationship from Figure 19 in topic 2.2.1.6. Since the relationship described is many-to-many, database design rules dictate that we build a new entity. This can be referred to as an associative entity because it reflects the many-to-many association between the objects. The middle section of Figure 38 shows the creation of the new *stockitem* entity and the adjustment of the relationships among the entities.

From this example, we see that a single object model relationship has been translated into multiple ERD relationships and has introduced a new entity. Careful analysis must be completed to accurately model in an ERD these object model relations.

Selecting and building associative entities is very common in database design. Usually, the process is straightforward and requires good understanding of the application business rules and database design basics. For our discussion, we do not go into the details of this process.

In some repsects, mapping the object relationships is similar to mapping basic entity model relationships. The association and aggregation relationship types are well known to many database designers. Discussion

here is not directed to the differences of object to entity relationship mapping, especially to the association and aggregation relationship types. Rather, we would like to note how very similar they are. Well-known data modeling and database design techniques apply quite adequately.

We have reviewed:

- Selecting a subsystem
- Identifying and mapping base object model classes to an ERD
- The types of relationships between the object model classes
- The multiplicity of these relationships
- Mapping an associative relationship from the object model to an ERD

Let's now look at the ERD model and reexamine it to be sure that we have captured the object model classes and relationships.

<u>New Entities, New Relationships, and the ERD</u>: We have now added several new entities to our ERD. These are *stock*, *denomination*, and the new associative entity *stockitem.* We have changed the relationship between stock and denomination. The many-to-many relationship is now one-to-many between both stock and stockitem and denomination and stockitem. We have completed, through example, the mapping of the associative class relationship in the object model to the ER model.

<u>Examine Relationship Types</u>: All models are subject to scrutiny. At this point it is the designer's role to scrutinize the ER model, review the object model, and ensure that the objects, classes, and the underlying relationships are directly reflected in the ERD. Refer to Figure 19 in topic 2.2.1.6, the initial OOA object model. We have not yet captured the relationship between cashier drawer, branch reserve, and stock in the ER diagram. This is a generalization relationship that implies inheritance and is somewhat like a supertype and subtype (although, in object modeling, generalization implies much more than a supertype and subtype relationship. It implies both attribute and behavioral inheritance).

Refer to Figure 39. Here the relationship between stock and cashier drawer and branch reserve needs close examination. For each branch, there are many cashier drawers and one branch reserve. Stock is made up of all the cashier drawers and the branch reserve stock. So, we can say that cashier drawer "is a" stock, and branch reserve "is a" stock. Hence, we have identified a generalization. Now how do we transfer this into the ERD?

PICTURE 39

Figure 39. Sample Implementation of Generalization

For accurate ER diagramming there is really only one choice for the representation. Identify each class (stock, cashier drawer, and branch reserve) as an entity. This conceptually captures the entities. ER diagramming can also use supertype and subtype relations. This also captures the data relationship. For physical implementation however, this does not do an adequate job. Let's review the implementation of this relationship to note some important points.

There are basically four choices for physically implementing a generalization relationship. We can identify:

- Each class as a table (stock, cashier drawer, branch reserve)
- Only the subclasses as tables (cashier drawer, branch reserve)
- Only the superclass as a table (stock)
- Each class as a table and the relationship as a table.

From a theoretical point of view, the first choice is an optimal choice. Each class is reflected as a table. Normalization rules can be followed easily, and we know there are model extensibility advantages found here. From an implementation view, we may have introduced additional complexity in data manipulation language (DML) for application developers.

From an implementation perspective, the second choice *may* be the best choice. However, there are disadvantages. Collapsing supertype attributes into subtypes can violate third normal form. Also, replicated data is introduced and can result in insert and update anomalies. Look for opportunities to collapse supertype attributes, but do consider it as a general rule implementation. Let's review an example.

See Figure 39, which shows how we modeled and then implemented the generalization relationship. Recall that stock was the superclass in the relationship. Stock, however, was never instantiated. The data resided in the subtypes or subclasses, the cashier drawer and the branch reserve. Due to these factors, we eliminated this table from the physical design. The attributes of stock were replicated into the subclass tables. For a review of the DDL, see Appendix B.

By implementing the relationship above we did break a data modeling design rule. The primary key of cashier drawer and the primary key of branch reserve serve as table identifiers *and* as identifiers of the subtype or subclass. Hence, we have added additional meaning to the composite primary key of the stockitem table. This could cause some problems, particularly in the extensibility of the preliminary design. However, for implementation elegance, the option was perceived as very attractive. As with all design decisions, there are tradeoffs. Our tradeoff here was either to add additional meaning to the keys of a table or implement a redundant entity. We chose the former.

The third choice, collapsing the subtypes into the supertype, is a poor implementation alternative. This will, most of the time, break normalization rules and provide insert, update, and delete anomalies. This could defeat entity modeling functions, data and relationship analysis, normalization, and domain integrity.

We have discussed mapping a generalization relationship to an ERD. We have also reviewed our physical implementation of this relationship. Generalization relationships are new to ER diagrammers and database implementers. If the relational data model and the implementation model will support object-oriented applications, it is particularily important to capture, model, and properly implement these relationships.

We have also reviewed options for mapping a generalization to an ERD. And, the implementation alternatives for generalization relationships and relational design have been discussed.

<u>Make Design Decisions</u>: We have noted, through example, our gernalization design decisions. Association and aggregation also pose some interesting and challenging design decisions that have been been given much attention in the academic and trade presses over the years.

Our design decisions are based on a small sample application. The application uses VisualAge, a true object-oriented application environment, as the sole database user. Security and performance have not been directly addressed in this test environment.

<u>ERD to Implementation Model</u>: This is the most difficult part of database design. For it is here that the tough implementation decisions are initially made, tested, and reviewed.

Converting an object model to an ER diagramming and to the subsequent database implementation requires design rigor. The differences between an object-oriented environment and a 3GL environment are many. With a relational database implementation, the major difference is in designing and implementing generalization relationships. Although the relationship is new to database adminstrators, the design requirements are the same: concurrency, security, data integrity, recoverability, and performance.

<u>Review Change</u>: Professionals in the information processing industry understand very well that change is a constant. This phase or step in design is to remind the database designers that object models change with business requirements. And, the capability to quickly reflect change in object-oriented applications provides the largest advantage for moving into this technology.

<u>Attribute Analysis</u>: Using attributes in 3GL applications and in OO applications is very similar. However, in an OO application, where attributes reside and how they are modeled can be quite different.

<u>How Do I Know I Have Done an Adequate Job?</u>: The crucial information required to determine design and implementation adequacy is the implementation-specific results. Testing, with sound test case development and test case feedback, is a crucial source of information. But as practitioners know, there is no better feedback data than production results. There is no "silver bullet" found in object-oriented applications with relational databases. Analysis, design, and testing rigor are required. Tools to complete these functions should become more prevalent in the market as more and more organizations begin adopting and

adapting to the object-oriented application development environment.

Appendix B lists the Data Definition Language for the tables of the sample application, and the REXX command files used in their creation.

<u>Database Access Design Approach</u>: With VisualAge it is quite easy to integrate database access with application logic. One drops a database query component into the composition editor, customizes the query, and connects the result table to the visual parts of the user interface.

<u>View-Data Application Architecture</u>: One can approach application development with a view-data architecture. This architecture suggests that user interface logic and data access logic are the primary parts of the application. This is well suited for decision support systems and simple application behavior implementations. VisualAge fully supports this type of application environment. Again, simple applications without extensive application logic requirements can fall into this view-data application development architecture.

<u>View-Model-Data Application Architecture</u>: In applications where business logic is dominant over data access logic, a view-data architecture is less appropriate. In this case, a view-model-data application architecture (see Figure 13 on page 37) leverages the full benefits of an object-oriented analysis and object-oriented design. Some of these benefits are:

- Dynamic real-world application model
- Dynamic application behavior analysis
- Application component reuse
- Application extensibility and adaptability.

<u>Object and Data Responsibility</u>: Objects consist of data (instance variables) and programs (methods). The object methods encapsulate the instance variables. Encapsulation requires the instance variables to be insulated from other objects. To alter the object's data, an object method must be invoked. This means that we must assign data access responsibility to our model objects.

Our object-oriented design approach enables integration of database access into an application prototype. With this approach application prototype code requires virtually no modification to integrate with database access. Let's examine this approach.

In Figure 40 we see that the application view is supported by test instance data. The application model contains logic to request and return access to the data. This data is initialized at application startup by the VisualAge environment. The view requests the data from the model and has no knowledge of the data access process. The view requests data and the model logic controls the data access.

PICTURE 40

Figure 40. View-Model and Test Instance Data

Figure 41 shows that the model's data access class issues requests for data from the VisualAge portfolio of wrapper classes.

Above, we noted that the view is isolated from data access. The model contains the data access class that requests data services. These services are provided by the VisualAge wrapper classes. Now, the model also becomes partially isolated from data access. The data access class is the only model object that provides and requests services from VisualAge.

PICTURE 41

```
+------------------------------------------------------------------------+
```

Figure 41. View and Model Isolation from Data Access

We assigned responsibilities to the objects in the object model data access class. The responsibilities of these objects are to:

- Know the table and row of all data items
- Apply changes to the table(s)
- Commit changes to the table(s).

Object Instantiation Responsibilities: Object data must be brought in memory (instantiated) from persistent storage before it can be used. The responsibility for bringing object data into memory is assigned following the knows-of hierarchy of the object model. For example, stock knows its stockItems. We assign stock the responsibility to bring stockItems into memory when needed. Here, sStock is the control object for the operations on stockItems.

Objects are brought in memory from persistent storage following different data policies. The control object knows and enforces this policy.

Class Hierarchy Design: We defined a persistent class for each model class in our hierarchy. This assists the application developer by keeping persistent and non-persistent object behavior separate and distinct. In this way different persistent object implementations can be used with minimal impact on overall application behavior.

Figure 42 depicts the inheritance hierarchy for the currency management subapplication.

```
+------------------------------------------------------------------------+
¦                                                                        ¦
¦                                                                        ¦
¦                                                                        ¦
¦                                                                        ¦
¦ PICTURE 42                                                             ¦
¦                                                                        ¦
¦                                                                        ¦
¦                                                                        ¦
+------------------------------------------------------------------------+
```

Figure 42. Persistent Classes Hierarchy

Client/Server Enabling: From an object technology view, Client/Server enabling is the process of refining the application's object design and distributing object functions for execution in a heterogeneous environment. This is accomplished by designing the interface between business application objects and the Client/Server wrappers provided by VisualAge.

Following are the required steps:

1. *Map objects to processors*, identify which objects each user will need to have instantiated on his or her machine.
2. Map *object replicas*, identify objects that need to be present at the same time in different images.
3. Design and build appropriate application or DBMS mechanism to keep replicas congruent from an application logic and database perspective.

Object Placement: Since Smalltalk cross image messaging is not available, replicas are needed today. DSOM promises to support cross image object messaging.

Keeping Object Replicas Synchronized: Managing different replicas of objects poses many problems. We can clasify these problems as logical problems and physical problems.

Logical problems: If we deal with a multiuser application, there may be a requirement for having an object replicated in all user images. The problem arises from the fact that each object has an identity, and at a given moment, each user may or may not have a copy of that object, affecting the requirements for object synchronization. We need to review the object model and use test cases to see which object is needed in the image to run the application. For instance, in the FCE application, logically I need MY cashier drawer and currency, but the currency lives outside my image. YOU need cashier drawer and currency, but currency lives outside YOUR image. Hence, currency requires logical replication in all images, while cashier drawer does not.

Physical problems: The physical distribution of objects is handled by

VisualAge through SOM/DSOM support. Today, VisualAge supports only a SOM client implementation. The full SOM/DSOM implementation will be available in the future in a phased manner.

Replicate Object Data Policies: Read policies are as follows (see Figure 43):

- Read data when object instantiated
- Read data when object queried
- Read data when object requested.

Write (update) policies are as follows:

- Update when updated
- Update when requested (foreground/background)
- Cannot update.

PICTURE 43

Figure 43. Overall View of Object Data Policy Analysis

The data policies for mapping objects to foreign data need to address two levels: the object level (where do we store an object as such) and an attribute level (where do we store an attribute for a given object). These two aspects of the policies are described in Figure 44 and Figure 45, respectively.

PICTURE 44

Figure 44. Individual Object Data Policy Analysis (Currency)

PICTURE 45

Figure 45. Individual Attribute Data Policy Analysis (Currency)

In a complex configuration environment it is good practice to build a matrix of data access per subsystem, by looking at each attribute of each object in each subsystem. This matrix is a very good tool to review data and message flows. It ensures consolidated subsystem selection and facilitates the normalization review for the design of the database. It is also useful for analyzing the patterns of data access and update.

To select among the data placement alternatives we can define for each object its data and function placement as follows:

Put data and process:

- Close to location of use (the locality of reference distributed database design rule can be applied to data and process placement)
- Where hardware support is reliable, redundant, scalable
- Where software will provide application performance and data integrity.

From a data view, object data policies give data element groupings, data reference patterns, and ideas on performance requirements. This helps the database administrator properly design the database.

A simple guideline for object method analysis is shown in Table 4.

Table 4. Object Method Placement Analysis. Chart for Client/Server function placement in a VisualAge Cli development environment.

Input	Process, Techniques, Tools	Deliverables
Selected subsystem	Reused functions	Function placement
Selected subsystem	Isolated functions	Function placement
Selected subsystem	Replicated functions	Function placement
Selected subsystem	Legacy code access	Function placement

Database integrity strategies

- Optimistic strategy: data contentions are very unlikely, each user can start his or her work trusting that nobody is going to use (update) the same data.

  Implementation: put a time stamp on tables and verify before updating that data is unchanged.
- Pessimistic strategy: data contentions are likely, so each user must get exclusive control of data before updating it.

  Implementation: flag the data as nonupdatable until the transaction is ended.

2.3.4.3 The System Architecture of the FCE Application

Figure 46 shows the system architecture of the FCE application.

```
+------------------------------------------------------------------------+
¦                                                                        ¦
¦                                                                        ¦
¦                                                                        ¦
¦                                                                        ¦
¦ PICTURE 46                                                             ¦
¦                                                                        ¦
¦                                                                        ¦
¦                                                                        ¦
+------------------------------------------------------------------------+
```
Figure 46. FCE System Architecture

2.4 Chapter 8. Sample Application: Design Work Products

This chapter illustrates aspects of the design of the sample application by showing work products of the phases of the application life cycle, including diagrams, models, and captures of key development screens.

Subtopics

2.4.1 Application Partition

Figure 47 shows the partitioning of the application into five subsystems or subapplications. All of the subapplications are developed with VisualAge. All persistent data accesses are through the Common Data Access subapplication. Applications developed with VisualAge require the VisualAge run-time environment. Figure 48 shows the high-level interactions among the five subapplications.

```
+--------------------------------------------------------------------------+
¦                                                                          ¦
¦                                                                          ¦
¦                                                                          ¦
¦                                                                          ¦
¦ PICTURE 47                                                               ¦
¦                                                                          ¦
¦                                                                          ¦
¦                                                                          ¦
+--------------------------------------------------------------------------+
```
Figure 47. Application Architecture

```
+--------------------------------------------------------------------------+
¦                                                                          ¦
¦                                                                          ¦
¦                                                                          ¦
¦                                                                          ¦
¦ PICTURE 48                                                               ¦
¦                                                                          ¦
¦                                                                          ¦
¦                                                                          ¦
+--------------------------------------------------------------------------+
```
Figure 48. Subapplication Interaction Diagram

2.4.2 Object Design Model

Figure 49 presents the refined object model that was used as the blueprint for designing the VisualAge public interfaces for each object.

```
+----------------------------------------------------------------------+
¦                                                                      ¦
¦                                                                      ¦
¦                                                                      ¦
¦                                                                      ¦
¦ PICTURE 49                                                           ¦
¦                                                                      ¦
¦                                                                      ¦
¦                                                                      ¦
+----------------------------------------------------------------------+
```
Figure 49. OO Object Design Model

2.4.3 Application Manager

Figure 50 shows the VisualAge Application Manager view of our project and the application and subapplications relationship. In this picture, FCEAForeignCurrencyExchange is our second iteration, and OOBAForeignCurrencyExchange is our third iteration.

```
+------------------------------------------------------------------------+
¦                                                                        ¦
¦                                                                        ¦
¦                                                                        ¦
¦                                                                        ¦
¦ PICTURE 50                                                             ¦
¦                                                                        ¦
¦                                                                        ¦
¦                                                                        ¦
+------------------------------------------------------------------------+
```
Figure 50. Application Manager

Figure 51 shows the VisualAge Application Browser view of the Currency Management Subapplication.

```
+------------------------------------------------------------------------+
¦                                                                        ¦
¦                                                                        ¦
¦                                                                        ¦
¦                                                                        ¦
¦ PICTURE 51                                                             ¦
¦                                                                        ¦
¦                                                                        ¦
¦                                                                        ¦
+------------------------------------------------------------------------+
```
Figure 51. Currency Management Subapplication Browser View

2.4.4 Test Strategy

Our development approach in each iteration can be characterized as both top-down and bottom-up. During analysis, it is important to take a top-down view for the application to be built. During design, however, we must build the application bottom-up--first construct the most primitive parts, and gradually assemble the composite parts of the application reusing those primitive parts built earlier.

As a consequence a bottom-up testing approach was also followed, that is, each component (part) was individually validated before its integration in an upper-level component.

Furthermore, during the unit test of each component, we basically treated each component as a "white box"; that is, to test the function of a component, we sometimes need to examine or look into the inside of the part. We even created a PartInspector to help us perform this task.

To perform integration test, we follow the use cases as test scenarios to validate how the parts will work together to provide the required functions. The individual parts can be viewed as "black boxes" during the integration test. Only their external behaviors are of interest to us in providing the expected function. And only when unexpected results are encountered do we go to the suspected components and examine them again, which is what we did during the unit test.

It is generally acknowledged that bottom-up testing is efficient, but it is often difficult in a traditional programming environment to provide test data to thoroughly verify the lower-level modules. In our environment we provided test data through the relations of the objects in the object model, making a collection of business objects available to be embedded in the components under development in order to verify their function.

<u>Encapsulate Test Data in a VisualAge Nonvisual Class</u>: Figure 52 shows the composition layout of the BankStubData class. The following explains our approach to devise the test data:

- The Bank class is embedded in a nonvisual component called BankStubData, which, as the name implies, represents the model containing the test data.

- The necessary business objects are created in the initialize method of the Bank class.

- External attributes of the Bank (customers, branches, currencies) are obtained through tear-off of the appropriate parts and add-to-interface.

- If external attributes are of the Collection class, a single test object is obtained by adding-to-interface the first element of the collection.

- The previous procedure (tear-off and add-to-interface) is repeated until the lowest level element is obtained.

The key benefit of this approach is that a single source of coherent and repeatable test data can be provided. For example, BankStudData can be added to the palette for reuse whenever testing is needed during the construction of the visual components. It allows the testing of the GUI before the databases are created. It facilitates the testing before we use the database query parts or build the persistent data access classes to access the relational database. All the nonvisual subcomponents in the BankStubData class are variables and have been added as external attributes to the BankStubData class. It becomes a convenient way of supplying test data during development of the application. Just add it as a subcomponent and connect the respective attributes.

BankStubData also provides a nice overview of the relations of the business objects of our object model, and a hierarchical structure of our data. It serves as a good chart to show and document the high level relationship among the different objects or classes.

As shown in Figure 52, a set of test data was created: The International OO Bank has a number of branches, including the San Jose Branch. The San Jose Branch has its branch stock and a number of cashiers were created as part of the test data. Customer orders were created for one of the cashiers.

The cashier has in his cashier drawer some stocked currencies. You can probably follow this description to figure out the rest of the test data scenarios.

The only drawback we found in our approach was that all team members

needed to update the Bank initialize method (which became a sort of bottleneck). But the team programming environment provides a strong control over concurrent updates which were well disciplined by VisualAge

```
+------------------------------------------------------------------------+
¦                                                                        ¦
¦                                                                        ¦
¦                                                                        ¦
¦                                                                        ¦
¦ PICTURE 52                                                             ¦
¦                                                                        ¦
¦                                                                        ¦
¦                                                                        ¦
+------------------------------------------------------------------------+
```

Figure 52. VisualAge Nonvisual Classes: Bank Tear-Off Attribute Diagram

2.4.5 Base Objects: VisualAge Nonvisual Classes

Base objects are those that are later used as building blocks of other composite objects. We have in our application base objects in both the visual and the non-visual class hierachies. Figure 53 shows the attributes of the class Bank which is a base object, as presented by the Public Interface Editor. The attributes and their corresponding classes are shown in Table 5, as well as an indication of whether or not each attribute is a derived attribute.

PICTURE 53

Figure 53. Base Object: Bank

Table 5. Base Nonvisual Class: Bank

Attributes	Instance Variable Class	Derived Attribu
customers	Dictionary	NO
currencies	Set	NO
currenciesList	OrderedCollection	YES
customersList	OrderedCollection	YES
name	String	NO
branches	Dictionary	NO
branchesList	OrderedCollection	YES
nextCustomerID	Number	NO

Derived attributes are derived from other basic attributes. They are normally read-only fields and do not need a set selector. Also, since these attributes do not have a set selector, you cannot use Quick Form to lay out the screen.

Figure 54 shows the attributes of the class Country. Table 6 describes the attributes of this class. The currency attribute is used to maintain the association of the Currency objects with the Country objects.

PICTURE 54

Figure 54. Base Object: Country

Table 6. Base Nonvisual Class: Country

Attributes	Instance Variable Class	Derived Attribu
currency	OOBCCurrency	NO
name	String	NO

Associations are bidirectional. We used a currency attribute in the Country class, and now we need a country attribute in the Currency class. This is shown in Figure 55 and in Table 7.

PICTURE 55

Figure 55. Base Object: Currency

Table 7. Base Nonvisual Class: Currency

Attributes	Instance Variable Class	Derived Attribu
country	OOBCCountry	NO
description	String	NO
short id	String	NO
exchange rate	Number	NO
denomination	Set	NO
denominationList	OrderedCollection	YES

The denominationList attribute is a derived attribute. The country attribute is of class OOBCCountry and is used to maintain the association with Currency.

DenominationType is a class that represents the unit of currency in a given country. Figure 56 and Table 8 show the characteristics of this class, which has a bitmap attribute of the abtBitmapDescriptor class for displaying the bitmap of the denomination, that is, show the sample of a HK$1000 bill. (In our example we use a butterfly as a fictional bill, as shown in Figure 57 in topic 2.4.7.)

PICTURE 56

Figure 56. Base Object: DenominationType

Table 8. Base Nonvisual Class: DenominationType

Attributes	Instance Variable Class	Derived Attribu
localCurrencyEquivalent	Number	YES
value	Number	NO
currency	OOBCCurrency	NO
description	String	YES
bitmap	AbtBitmapDescriptor	NO

2.4.6 Nonvisual Classes Hierarchy

VisualAge allows you to inherit from any Smalltalk class when you create a nonvisual class. The default is set to be a subclass of AbtAppBldPart. Most of our base nonvisual classes are created as subclasses of AbtAppBldPart or of their abstract classes. For example, CashierStock was created as a subclass to Stock. The nonvisual classes hierarchy is as follows:

```
Object
   AbtObservableObject
     AbtPart
        AbtCompositePart
           AbtAppBldPart
              AbtAppBldrView...
              ............
              OOBCAccount
              OOBCBank
              OOBCBankStubData
              OOBCBranch
              OOBCCashier
              OOBCCountry
              OOBCCurrency
              OOBCCustomer
              OOBCDenominationType
                 OOBCBillType
                 OOBCTCheckType
              OOBCOrder
                 OOBCCustomerOrder
                 OOBCBranchOrder
              OOBCOrderItem
              OOBCPersistentObject
              OOBCStock
                 OOBCBranchReserve
                 OOBCCashierDrawer
              OOBCStockItem
              OOBCStockToOrderTransfer
              OOBCUSAddress
              ..............
```

Note: We have prefixed application with OOBA, subapplication with OOBS, and class with OOBC.

2.4.7 Base Visual Classes

A sample of visual classes is presented here. We will use the CurrencyManagement subsystem as an example.

The attributes of these visual classes are implemented by using the VisualAge variable part and adding it to the public interface.

Figure 57 shows a primary view for denominationType. It is used to display, in this case, a picture of the denomination. It has the following external interfaces and connections:

- Attribute

  - The Denomination Type - an OOBCDenominationType variable instance

- Connections

  - Bitmap of the Denomination Type <--> graphicsDescriptor of Label

  - Description of the Denomination Type <--> labelString of Label.

```
+------------------------------------------------------------------------+
¦                                                                        ¦
¦                                                                        ¦
¦                                                                        ¦
¦ PICTURE 57                                                             ¦
¦                                                                        ¦
¦                                                                        ¦
+------------------------------------------------------------------------+
```
Figure 57. Base Visual Class: DenominationType View

Figure 58 shows the composite view of denominationList. It has a public attribute variable instance called the DenominationList. We reuse the denominationTypeView components shown in Figure 57. The selectedRowObject attribute of the table is connected to the denominationType attribute of the denominationTypeView subcomponent. Therefore, when the menuItem Open is selected, it will open the denominationType view to show the picture of the selected currency.

```
+------------------------------------------------------------------------+
¦                                                                        ¦
¦                                                                        ¦
¦                                                                        ¦
¦ PICTURE 58                                                             ¦
¦                                                                        ¦
¦                                                                        ¦
+------------------------------------------------------------------------+
```
Figure 58. Base Visual Class: DenominationListTypeView

The denominationListTypeView component has the following external interfaces and connections:

- Attribute

  - The Denomination List - an OrderedCollection variable instance

- Connections

  - Self of menu <--> menu of Table

  - Self of the Denomination List <--> rows of Table

  - The denominationType of denominationTypeView <--> selectedRowObject of Table.

  - Clicked on Open push button ---> openWidget of denominationTypeView

Figure 59 shows the composite view of currency. It has a public attribute variable instance of currency. The country instance variable is created by tearing off from the Currency. We reuse the denominationListTypeView shown in Figure 58 as a subcomponent.

```
+------------------------------------------------------------------------+
¦                                                                        ¦
¦                                                                        ¦
```

```
¦                                                                         ¦
¦                                                                         ¦
¦ PICTURE 59                                                              ¦
¦                                                                         ¦
¦                                                                         ¦
¦                                                                         ¦
+-------------------------------------------------------------------------+
```
Figure 59. Base Visual Class: CurrencyView

Figure 60 shows the composite view of currency create view. This view consists of a reusable component, the currencyView, shown in Figure 59 as a subcomponent. We also use a deferred update component for editing the currency.

```
+-------------------------------------------------------------------------+
¦                                                                         ¦
¦                                                                         ¦
¦                                                                         ¦
¦                                                                         ¦
¦ PICTURE 60                                                              ¦
¦                                                                         ¦
¦                                                                         ¦
¦                                                                         ¦
+-------------------------------------------------------------------------+
```
Figure 60. Base Visual Class: CurrencyCreateView

The public interfaces for this view are:

1. Attribute the Currency

2. Event newCurrencyCreated.

We use an event-to-script connection to signal the newCurrencyCreated event when the Add push button is clicked. The method of signalAddButtonPushed is shown below:

```
  signalAddButtonPushed

  &carret.self signalEvent: #newCurrencyCreated.
```

Figure 61 shows the composite view of currency list view. It has a public interface attribute variable named the Bank. The currenciesList of the Bank is created through tearing off the Bank from public interface attribute variable.

```
+-------------------------------------------------------------------------+
¦                                                                         ¦
¦                                                                         ¦
¦                                                                         ¦
¦                                                                         ¦
¦ PICTURE 61                                                              ¦
¦                                                                         ¦
¦                                                                         ¦
¦                                                                         ¦
+-------------------------------------------------------------------------+
```
Figure 61. Base Visual Class: CurrencyListView

We reuse the currencyEditView component shown on Figure 62. It has a menu, four push buttons and a CurrencyEditView subcomponent. The selectedRowObject attribute from the table is connected to the Currency attribute of the currencyEditView subcomponent. Therefore, when the open menuItem or the open push button is clicked, it will open the CurrencyEditView component to show the edit view of the selected currency.

Figure 62 shows the composite view of currency edit view. This is the first view that has a primary window, which is one of the window defined by the user in the analyst prototype. As an example to show how to use subclassing, we will create this view by subclassing the CurrencyCreateView. The differences between the two views are:

- The edit window has a Change button instead of an Add button

- The edit window has a primary window

- The event is changeCurrencyRequested instead of newCurrencyCreated.

```
+------------------------------------------------------------------------+
¦                                                                        ¦
¦                                                                        ¦
¦                                                                        ¦
¦                                                                        ¦
¦ PICTURE 62                                                             ¦
¦                                                                        ¦
¦                                                                        ¦
¦                                                                        ¦
+------------------------------------------------------------------------+
```

Figure 62. Base Visual Class: CurrencyEditView

After the visual class is created, a copy of the CurrencyCreateView is also created and can be seen on the composition editor. Then:

1. Add the window shell as the primary window.

2. Add an event changeCurrencyRequested to the public interface.

3. Create the signalAddButtonPushed method to signal the new event symbol.

4. Change the text of the Add button to Change.

Figure 63 shows the Currency Management System main window. We reuse the CurrencyCreateView shown on Figure 60 here. The Currency attribute variable of the CurrencyCreateView is connected to the template attribute of the Currency Template. (This is implemented using our Object Cloner class.)

```
+------------------------------------------------------------------------+
¦                                                                        ¦
¦                                                                        ¦
¦                                                                        ¦
¦                                                                        ¦
¦ PICTURE 63                                                             ¦
¦                                                                        ¦
¦                                                                        ¦
¦                                                                        ¦
+------------------------------------------------------------------------+
```

Figure 63. Currency Management: a new Currency

The newCurrencyCreated event is connected to the action named clone of the Currency Template and to the add action of the currency collection. Hence, when the Add push button is clicked a new currency instance will be created and added to the currency of the Bank attribute (an orderedCollection).

Figure 64 shows the Currency List notebook page. We reuse the CurrencyListView shown on Figure 61 here. The currenciesList of the Bank attribute variable of CurrencyListView is connected to the currency of the Bank attribute so that a list of currencies can be displayed in the list.

```
+------------------------------------------------------------------------+
¦                                                                        ¦
¦                                                                        ¦
¦                                                                        ¦
¦                                                                        ¦
¦ PICTURE 64                                                             ¦
¦                                                                        ¦
¦                                                                        ¦
¦                                                                        ¦
+------------------------------------------------------------------------+
```

Figure 64. Currency Management: List of Currencies

2.4.8 Visual Classes Hierarchy

The visual classes hierarchy is as follows:

```
 Object
    AbtObservableObject
      AbtPart
         AbtCompositePart
            AbtAppBldPart
               AbtAppBldrView
                  .........
                  OOBCInternationalOOBankView
                  OOBCAccountView
                  OOBCBankNameView
                  OOBCBankStubData
                  OOBCBranchNameView
                  OOBCCashierView
                  OOBCCashierNameView
                  OOBCCountryView
                  OOBCCurrencyView
                  OOBCCurrencyCreateView
                     OOBCCurrencyEditView
                  OOBCCurrencyListView
                  OOBCCustomerView
                     OOBCCustomerEditView
                     OOBCCustomerCreateView
                  OOBCCustomerManagementView
                  OOBCCustomerNameView
                  OOBCDenominationTypeView
                  OOBCDenominationTypeListView
                  OOBCDenominationTypeEditView
                  OOBCCustomerOrderListView
                  OOBCCustomerOrderNumberDateView
                  OOBCCustomerOrderView
                  OOBCBranchOrderView
                  OOBCOrderItemListView
                  OOBCOrderManagementView
                  OOBCPersistentObject
                  OOBCStockView
                  OOBCStockCurrencyAvailablityView
                  OOBCStockItemView
                  OOBCStockItemEditView
                  OOBCStockItemListView
                  OOBCStockItemListEditView
                  OOBCStockManagementView
                  OOBCStockToOrderTransfer
                  OOBCSystemLogonView
                  OOBCUSAddressView
                  -------------
```

2.4.9 Examples from the FCE Application

The screen captures in this section reflect several aspects of the Currency subsystem of the FCE application.

Subtopics

2.4.9.1 Run-time-Screen captures

The first screen seen by the user when starting the application is shown in Figure 65.

PICTURE 65

Figure 65. International OO Bank Main Screen

After clicking on the eye icon of the main screen, the user is presented with the VisualAge logon screen, as shown in Figure 66.

PICTURE 66

Figure 66. System Logon View

After logging on to VisualAge, the user is presented again with the main screen where there are four icons that can be selected to start the respective application. Assuming the user selected the Currency icon, the program displays the screen shown in Figure 67.

PICTURE 67

Figure 67. Currency Management System: Run Time

In our example, the user selects now the second page of the notebook shown in the CurrencyManagement System screen, that is, the list of the countries with their respective currencies (see Figure 68).

PICTURE 68

Figure 68. Currency Management: Country List

The currency application allows the user to display an image of a given denomination type, that is, a bill of a valid currency of a certain value. Figure 69 shows the display of an imaginary denomination type.

PICTURE 69

Figure 69. Denomination Type Display

The screen captures below illustrate some aspects of other FCE applications. For instance, Figure 70 shows the display of the money

available at a certain moment in a cashier drawer.

```
+------------------------------------------------------------------------+
¦                                                                        ¦
¦ PICTURE 70                                                             ¦
¦                                                                        ¦
+------------------------------------------------------------------------+
```
Figure 70. Stock Management: Cashier Drawer

When selecting the notebook entry "Details" of the cashier drawer screen, the user is presented with the screen shown in Figure 71.

```
+------------------------------------------------------------------------+
¦                                                                        ¦
¦ PICTURE 71                                                             ¦
¦                                                                        ¦
+------------------------------------------------------------------------+
```
Figure 71. Cashier Drawer Details

Figure 72 displays the money the cashier drawer has in excess of a given currency and that has to be transferred to the bank's reserve.

```
+------------------------------------------------------------------------+
¦                                                                        ¦
¦ PICTURE 72                                                             ¦
¦                                                                        ¦
+------------------------------------------------------------------------+
```
Figure 72. Transfer of Currency to Reserve

Figure 73 displays the money in the cashier drawer that has reached a value below normal for a given currency and has to be replenished from the bank's reserve.

```
+------------------------------------------------------------------------+
¦                                                                        ¦
¦ PICTURE 73                                                             ¦
¦                                                                        ¦
+------------------------------------------------------------------------+
```
Figure 73. Replenishment of Currency from Reserve

2.4.9.2 Development Time Screen Captures

The figures in this section show how we built some of the parts of the stock management application. They also illustrate how an application can be built from basic or previously developed components.

Figure 74 shows the cashier drawer main screen. The notebook is connected to the nonvisual objects that represent the drawer and the bank stock, respectively.

```
+------------------------------------------------------------------------+
¦                                                                        ¦
¦ PICTURE 74                                                             ¦
¦                                                                        ¦
+------------------------------------------------------------------------+
```

Figure 74. The Cashier Drawer Development Screen

Figure 75 shows the excess currency page of the cashier drawer notebook. The Transfer to Reserve button is connected to both the drawer object and the branchStock object.

```
+------------------------------------------------------------------------+
¦                                                                        ¦
¦ PICTURE 75                                                             ¦
¦                                                                        ¦
+------------------------------------------------------------------------+
```

Figure 75. Cashier Drawer Details Screen

Figure 76 shows the replenish currency page of the cashier drawer notebook. The Replenish from Reserve button is connected to both the drawer object and the branchStock object.

```
+------------------------------------------------------------------------+
¦                                                                        ¦
¦ PICTURE 76                                                             ¦
¦                                                                        ¦
+------------------------------------------------------------------------+
```

Figure 76. Replenish from Reserve

The figures below describe aspects of the development of the StockItem model class, together with its respective view classes. The StockItem class represents the items in the bank's stock.

Figure 77 shows the Public Interface Editor showing the generation of the get and set methods for the stock instance variable of the StockItem class.

```
+------------------------------------------------------------------------+
¦                                                                        ¦
¦ PICTURE 77                                                             ¦
¦                                                                        ¦
+------------------------------------------------------------------------+
```

Figure 77. Stock Management System

The figures that follow show the building from parts mechanism: Figure 78 is built from the part shown in Figure 79, which is made of the parts shown in Figure 80 and Figure 81, respectively.

```
+------------------------------------------------------------------------+
¦                                                                        ¦
¦ PICTURE 78                                                             ¦
```

```
¦                                                                     ¦
¦                                                                     ¦
¦                                                                     ¦
+---------------------------------------------------------------------+
```
Figure 78. StockItem Edit View

```
+---------------------------------------------------------------------+
¦                                                                     ¦
¦                                                                     ¦
¦                                                                     ¦
¦ PICTURE 79                                                          ¦
¦                                                                     ¦
¦                                                                     ¦
+---------------------------------------------------------------------+
```
Figure 79. StockItem List Edit View

```
+---------------------------------------------------------------------+
¦                                                                     ¦
¦                                                                     ¦
¦                                                                     ¦
¦ PICTURE 80                                                          ¦
¦                                                                     ¦
¦                                                                     ¦
+---------------------------------------------------------------------+
```
Figure 80. StockItem List View Component

```
+---------------------------------------------------------------------+
¦                                                                     ¦
¦                                                                     ¦
¦                                                                     ¦
¦ PICTURE 81                                                          ¦
¦                                                                     ¦
¦                                                                     ¦
+---------------------------------------------------------------------+
```
Figure 81. StockItem View Component

2.5 Chapter 9. Recommendations

In this chapter we provide recommendations based on the experience we gained from our residency project. Although we try to focus on items specific to VisualAge, you may find some of our suggestions generally applicable to any object-oriented projects.

Subtopics

2.5.1 Planning Considerations for a VisualAge Development Project

When starting an application development project with VisualAge, several issues have to be considered to lead the project to a successful conclusion in a productive way. In this section we describe these issues from a project leader's perspective.

Subtopics

2.5.1.1 Skills Required

VisualAge is a very powerful tool that makes the construction of applications from parts relatively easy. Applications can be built visually with VisualAge from existing components without in-depth object-oriented knowledge or Smalltalk skills. On the other hand, building new parts requires skills in the techniques of object-oriented design and Smalltalk programming.

While VisualAge provides a robust framework with a rich set of general purpose components, such as database query and access, and communications support, there may not be many domains or application components available to your project. A project should be prepared to construct its own application components and hence will require skilled Smalltalk programmers with object-oriented design experience.

VisualAge cannot totally eliminate the need to write Smalltalk code for a project. A rule of thumb is that 20% or more of your code will still be Smalltalk coding. Therefore, the project team should include developers skilled in object-oriented analysis, design, and Smalltalk programming.

How much Smalltalk programming knowledge is needed? Following are the various aspects in adopting Smalltalk as an object-oriented programming language:

- Syntax: Smalltalk syntax is simple and can be quickly learned by programmers.

- Class hierarchy: Smalltalk comes with a rich set of classes, and a good Smalltalk programmer can leverage this reuse instead of writing code. However, only a handful of classes (Collection, Magnitude) are absolutely needed to start coding, and the others can be learned later as the programmer has gained more knowledge about the basic classes.

- Environment: some Smalltalk facilities should be mastered, for example, Debugger, Inspector, and Browser.

- Paradigm: The Smalltalk object-oriented programming paradigm should be well understood.

2.5.1.2 Iterative Development Process

The number of iterations required to produce a good working application depends on the object-oriented skills of the development team. However, we feel that a minimum of three iterations is required to deliver an acceptable solution. We suggest the various focuses of each iteration later in this section.

It is worth spending some time in planning the first iteration and defining the output required. As skills and knowledge about the problem domain increase, the time required for subsequent iterations is reduced. As a rule of thumb, the relative time spent on each of the three iterations is expected to be in a ratio of 2.0 : 1.5 : 1.0.

Define completion criteria for each iteration before start. This will help to manage the iterations and avoid confusion about when an iteration is done and when to start a new iteration.

Use an iterative approach for detail design and construction within each iteration; that is, fully implement and test each component, before going on to the next one. Test the primitive components thoroughly as they will be used by other composite components.

Be prepared to reengineer your prototype from earlier iterations.

2.5.1.3 Project Team Size and Organization

Divide the project into "manageable" subproject teams. Each team can then be responsible for the development of an "application" under the VisualAge team programming environment. And each "sub-application" can be assigned to one or two developers. Each team should normally consist of no more than six developers. There is a high degree of communication and discussion among the developers in an object-oriented VisualAge development projects. Small teams will help to keep design in sync among the small number of designers working on an application or subapplication.

There is a need to assign a single project focal point to manage the versioning and configuration of the application(s) being developed in the VisualAge library.

2.5.1.4 Methodology and CASE Tools

Adopt one "backbone" methodology for the project, and stick to its notation throughout the whole development lifecycle. See "Which Object-Oriented Methodology to Use?" in topic 2.1.4.2. Apply techniques that work for you, for example, try the use-cases technique to get the correct requirements, and CRC for class responsibility analysis.

Reconcile to common semantics for the terminology used among the team members early in the project. Try to use the same vocabulary to avoid confusion.

Object-oriented CASE tools are helpful, mostly for documentation purposes. This is because most object-oriented modeling CASE tools do not provide dynamic integration or integrity check while going from analysis to design, and then to implementation. Furthermore, code generation is provided by VisualAge, so there is no direct linkage to generate code from the design work product kept in the CASE tool's library.

<u>A simple application as a learning and exploratory pilot project</u>: It is worthwhile investing in a pilot project as a way of learning by doing.

2.5.1.5 Team Programming Environment: Pragmatic Tips

To help locate the classes associated with an application and to ease the migration to new application names, we suggest set up the following naming conventions:

- Naming convention: XXXYzzzzzzzzzzzz

  Name the application with a three-character prefix, followed by one character for application, subapplication, and class indicator, followed by the fully qualified application or class name:

  1. XXX = application char description
  2. Y = A | S | C appl or subappl or class
  3. z=applname fully qualified name.

  For example, "OOBAForeignCurrencyExchange" represents the application name, and "OOBCStock" represents a class name within the application.

2.5.1.6 Pragmatic Design and Style Guidelines for VisualAge

The following rules of thumb are suggested to keep the design simple and manageable:

- The number of classes under a subapplication should be within the range of 30 to 50.

- A composite part should contain no more than 5 to 8 subparts.

- The code for a method should usually not be larger than one page.

- Select subapplications based on the subsystems from your object model.

- Try to keep the number of connections within a Composition Editor screen under a dozen or so.

- Avoid "code hook" wherever possible.

Some suggestions from Larry Smith's append in VISBETA forum:

- Name nonvisual parts the same as their class names.

- Document the composition using the comments section of the script editor for the class of the visual part. Short comments on methods should be in the method body, while long descriptions for methods should be in the comments section.

- Use forms whenever possible, especially for notebook pages. Forms are abstractions that can greatly simplify a composition.

- If a class is referenced by another part in a composition, put a part representing that class on the work surface. For example, put a nonvisual part representing the instance class on the worksurface next to an object factory that generates that class, and put a nonvisual part representing the class of the elements of an ordered collection next to the ordered collection. This provides a little bit more documentation and also makes it easy to browse those referenced classes as needed.

- Position connection lines to avoid crossing lines whenever possible.

- Disable any control (so it is greyed out) that has not yet been implemented.

- All buttons in a group should generally be the same size.

  For example, the OK, Help, and Cancel buttons should be the same size, while the Calculate and Refresh button can be a little bigger.

- Align controls horizontally and vertically as much as possible.

2.5.1.7 Tips on Using VisualAge

The following are some useful considerations on VisualAge usage:

- File-out/file-in is not the recommended way to move things between team edition libraries. Use import/export instead. If you can see the other library, you can export directly into it. If not, create a new, empty library (Transcript-System-Create new library) and export to the new library, send the new library to the site that has the destination library, and import into the destination library from the new library. Howoften you generate code is really up to you. Generating on every save is not necessary, unless it makes you feel more secure and you do not mind the time. You might generate before every build as a compromise. Remember that generating code is not necessary for the normal functioning of the system or application. It is useful for backup and recovery and for exporting and importing from the single user version (when it becomes available).

- Make sure you have the 'initialize' instance method in your nonvisual class to instantiate any attributes that do not belong to the standard Smalltalk classes.

2.5.2 Advice for Component Builders

The following advice is adapted from :bibref refid=kin94.. The purpose is to guide application developers (programmers) to build components that are intended to be used by other applications. The guidance hopefully will ensure that the components are properly built to allow other developers to rapidly assemble the reusable components into viable applications.

Subtopics

2.5.2.1 Designing a Good Public Interface

The goal when designing the public interface is to make the part easy and natural to use with visual programming techniques. It is likely that your design will start out requiring too many connections: each button had to be individually connected to some primitive part's action. It is advisable to evolve the design to simplify the connections you need. You should be cautious about requiring application assemblers to make a large number of connections, connections with lots of parameters, or connections that must be made in a specific order. Each of these conditions is hard to understand at a glance in the Composition Editor, so they should be avoided.

A rule of thumb is that there should not be more than 12 or so connections in one Composition Editor screen.

2.5.2.2 Building Visual Components (Parts)

If you are building visual parts, like the button blocks and individual buttons, you can probably do most of the work in the Composition Editor. The button blocks are just forms that contain individual buttons. The size and position settings of each individual button are such that the button maintains the proper fraction of the form's total area. If the button block is moved or resized, the buttons adjust themselves as necessary. The only hand-written code for the blocks is the code that signals events for each individual button click. This is not strictly necessary, but it makes the blocks much more useful than they would otherwise be. The individual buttons contain all the intelligence needed to initiate the play, stop, record, and other actions.

They override certain inherited methods so that when they are clicked, they check to see if a connection to a player exists, and if it does, they send the appropriate action message to the player.

2.5.2.3 Providing Examples

No matter how simple and obvious your part's public interface seems, be sure to provide plenty of examples showing creative ways that the part can be used. When you test your parts, be sure to involve people who do not have preconceived notions about how the parts should be used. Since visual programming is so quick and easy, they are certain to try things that you did not anticipate.

2.5.2.4 Portability and Performance

Remember that VisualAge is a cross-platform tool. As you write the Smalltalk code that implements your public interface features, you are responsible for keeping your code portable. Drawing on the class hierarchy of VisualAge itself will help in this task.

When you write methods that manage your attributes, remember that attribute-to- attribute connections may cause the attribute value to be requested many times, so the methods that return them should be quick. Similarly, the methods that implement your actions should be quick enough to keep the user interface responsive and should use separate threads when necessary.

2.5.3 Useful Development Support Components

We have developed the following development support components which we found useful for our residency project:

- Test Data Stub

  While you are creating the nonvisual classes for your project, develop some sample data alongside. It proved to be very useful in our case to test the GUI before we built the persistent data access component. See Figure 52 in topic 2.4.4 for an example of our test data.

- Components Inspector

  This is a very simple component that we created for debugging and inspecting any object during development. The idea is very simple. The component has one attribute, componentToBeInspected, and an action, inspect. You can inspect any of the subcomponents during your composition editor session by connecting any attribute of any subcomponent to the componentToBeInspected attribute and connecting the inspect action to a push-button-clicked event. It is a very simple and easy way of inspecting the value of an attribute at run time.

- Object Cloner - Object Factory - Pros and Cons

  The limitation of the Object Factory class is that it does not copy the contents of any subattributes from the source object. For example, Customer class has an attribute called Address of class USAddress. If you create two instances of Customer using the Object Factory component, the Address attribute in both Customer instances points to the same instance of Address. We have created a new component called Object Cloner, which has two external attributes, template and newObject, and one action, clone. The clone method just returns a deepCopy of the template to the newObject. This resolves the limitation of the Object Factory. The clone method is defined as:

  ```
  clone

  ^self newObject: template deepCopy
  ```

- Additional components that should help enhance the visual programming capability are:

  - do: Iterator
  - select: Iterator
  - reject: Iterator
  - collect: Iterator
  - data type converter (for example, convert a Dictionary to an OrderedCollection)
  - conditional (branch) Selector.

A.0 Appendix A. Requirements Specifications

This appendix describes the specifications of the Foreign Currency Exchange application.

Subtopics

A.1 Branch Functions

Because smaller branches do not maintain their own stocks of foreign currency and travelers' checks, they can satisfy demand only by ordering from the center on the customer's behalf. Larger branches do maintain stocks, and customer sales and purchases are handled by cashiers allocated to a foreign currency and travelers' check "bureau" within the branch.

Functions performed at the branch include:

- Customer order management

  - Customer purchases from branch cashier

    This is the most frequently performed function in this application. Purchases are normally for one currency and one check for the destination country. Payment for this service can be by cash, credit card, local check, or a debit to the customer's account.

    Stock levels are reduced, a customer tab is printed, and accounting entries are passed to the accounts application.

    Other facilities of this function are:

    -- Check stock levels

    -- Reduce branch stock level

    -- Determine currency and/or check denominations (small, mix, large, or specified)

    -- Obtain exchange rate

    -- Print tab (duplicate for signature)

    -- Generate accounting entries (debit currency code and credit dollars branch account)

    -- Handle multiple currencies

    -- Handle multiple currencies and checks

    -- Handle country restrictions, warnings, general information.

  - Customer sells to branch cashier

    Note: This function is not implemented in the application.

    Tourists and travelers returning with excess currency and checks sell or cash in currency and checks to the bank. Notes and checks are checked for forgeries by reference to textual information on legitimate denominations, descriptions, and known forgery defects. Other activities in this function are:

    -- Obtain exchange rates

    -- Increase branch stock level

    -- Print tab (duplicate for signature)

    -- Handle multiple currencies

    -- Handle multiple currencies and checks

    -- Pass accounting entries (debit currency code(s) and credit dollars branch account)

  - Customer order form that cannot be satisfied.

    Note: This function is not implemented in the application.

    Not all branches stock currencies and checks. Very few branches stock the full range as this would not be economical. All orders that cannot be met by the branch are routed to the center. If payment has been made or the customer has an account with the bank, the currency or checks will be sent to the customer's address; otherwise it will be sent to the branch for later collection. Other activities in this function are:

    -- Handle country restrictions/warnings/general information

    -- Handle currency and/or check denominations (small, mix, large, or specified)

-- Handle multiple currencies

-- Handle multiple currencies and checks

-- Obtain exchange rate

-- Take deposit if noncustomer

-- Print tab (duplicate for signature)

-- Pass accounting entries (debit currency code and credit dollars branch account)

- Cashier management

  - Cashier stock reconciliation

    At the end of each day, or more frequently, each cashier must verify that the checks and foreign currency in his or her cabinet are equal to the totals held in the system. This is done by viewing the values held for each denomination held, within each currency or check.

    If these totals cannot reconcile, the discrepancy is passed to an Overs and Shorts account and the totals are amended accordingly.

    Authority can be granted only by a supervisor.

    Activities include:

    -- Obtain exchange rate for each currency or check

    -- Display each currency or check totals and local currency equivalent

    -- Display each currency or check denomination totals and local currency equivalent

    -- Order replenishment stock if minimum stock quantity reached

    -- Send excess stock to center if maximum stock quantity exceeded

    -- Display total local currency equivalent

    -- Raise compensating accounting entries for small losses or gains

    -- Archive reconciliation.

- Branch management

  Note: The branch management functions are not implemented in the application.

  - Branch stock replenishment

    At the end of each day, the requests of the cashiers are consolidated. Each request can be either an order to replenish stock or to send excess stock. A consolidated branch order is then sent to the center.

  - General inquiry of branch stocks

    Similar to customer order (branch) but no update intent

  - General inquiry of central stocks

    Similar to customer order (center) but no update intent

  - Forgery recognition (computer image)

    Inquiry only (compare image to real note and verify descriptive information on what faults to look for).

A.2 Center Functions

The center is responsible for supplying foreign currency and traveler's checks to the branches (outlets) and for selling off excess foreign currency received from branches. They do this by dealing on the foreign currency markets, arranging bulk shipments at favorable exchange rates.

Functions performed at the center include:

- Bank management

  Note: The bank management functions are not implemented in the application.

  - Branch order

    Orders from branches are normally for several currencies and checks. All orders are processed in batch mode, bulk quantities are picked, delivery arrangements are made, postal charges are set, stock levels are reduced and accounting entries are passed between branch and the central accounts.

    Branch order functions include:

    -- Currency and/or check denominations (small, mix, large, specified).

    -- Check stock levels

      - Single currency
      - Single checks
      - Multiple currencies
      - Multiple checks
      - Multiple currencies and checks

    -- Print picking lists

    -- Reduce center stock level

    -- Obtain exchange rate

    -- Weigh packages and establish postal charges

    -- Print branch documents

    -- Pass accounting entries (debit currency code credit dollars center account).

  - Customer order

    Where branches have been unable to satisfy any part of the customer order, the whole order is supplied by the center. Purchases are normally for one currency and one check for the destination country.

    If payment for this service has been made at the branch, then the foreign currency and travelers' checks will be mailed to the customer; otherwise, it will be mailed to the branch for collection.

    Stock levels are reduced, a customer tab is printed, and accounting entries are passed to the accounts application.

    Customer order functions include:

    -- Handle country restrictions/warnings/general information

    -- Handle currency/check denominations (small, mix, large, or specified)

    -- Handle check stock levels

    -- Handle single currency

    -- Handle single checks

    -- Handle multiple currencies

    -- Handle multiple checks

-- Handle multiple currencies and checks

-- Print picking lists

-- Reduce center stock level

-- Weigh packages and establish postal charges

-- Obtain exchange rate

-- Print customer tab

-- Pass accounting entries (debit currency code credit dollars center account)

- Branch excess

Excess foreign currency received from the branches is counted, reconciled with branch delivery record, and added to central stocks. Where the customer has not accepted an order sent by the center to the branch for collection, then reversing accounting entries for this cancellation or return is passed.

Customer order functions include:

-- Add to central stock

-- Match value to branch file

-- Pass accounting entries

-- Reverse entry for cancellations/returns.

- Central stock reconciliation

At the end of each day, or more frequently, the center verifies that the checks and foreign currency in its stocks are equal to the totals held in the system. This is done by viewing the values held for each denomination held, within each currency or check and counting the actual stock.

If these totals cannot reconcile, the discrepancy is passed to an Overs and Shorts account, and the totals are amended accordingly.

Authority can only be granted by the senior manager.

Activities include:

-- Obtain exchange rate for each currency and/or check

-- Display each currency and/or check totals and local currency equivalent

-- Display each currency/check denomination totals and local currency equivalent

-- Order replenishment, if minimum stock quantity reached, from other banks (U.S.A. and abroad) through the foreign note dealers

-- Sell excess, if maximum stock quantity is exceeded, to other banks (U.S.A. and abroad) through the foreign note dealers

-- Display total local currency equivalent

-- Raise compensating accounting entries for small losses/gains

-- Archive reconciliation.

- Maintain branch stock limits

Periodically the stock held at each branch is reviewed to check whether the stock minimum and maximum levels are still appropriate. A number of "what if" conditions are used to establish a revised set of limits, including seasonal, period on period, and general demand conditions.

Stock limit processing includes:

-- General inquiry stock levels still valid?

   - Season change, period on period demand change, abnormal condition, and so forth

- Change stock level minimum or maximum
- Add or remove stock type or denomination.

- Maintain center stock limits

  Similar to branch but include issues of bulk transport, international availability, and capacity.

- Maintain exchange rates

  Dealers maintain the rates for each currency and check by comparing with other banks' rates, general market rates from Reuters, telerate, and the like, and by checking general availability.

  A different rate may be applied for small and large denominations.

- Miscellaneous transactions

  - Maintain forgery images
  - Create, amend, delete currency images
  - Produce customer labels, envelopes, documents
  - Produce branch sack labels and documents.

B.0 Appendix B. Database Definition

This appendix lists the Data Definition Languages that define the tables used in the sample applications, and the REXX command files used for their creation.

Subtopics

B.1 Data Definition Language for Relational Tables

ACCOUNT Table

```
  CREATE TABLE ACCOUNT
         (ACCT_ID                  CHAR(9)      NOT NULL,
          ACCT_TYPE                CHAR(14)     NOT NULL,
          ACCT_BALANCE             FLOAT        NOT NULL,
          ACCT_BRID                CHAR(4)      NOT NULL,
          PRIMARY KEY (ACCT_ID),
          FOREIGN KEY ACCTBRCH (ACCT_BRID)
              REFERENCES BRANCH ON DELETE RESTRICT)
```

BRANCH Table

```
  CREATE TABLE BRANCH
         (BRCH_ID                  CHAR(4)      NOT NULL,
          BRCH_NAME                CHAR(20)     NOT NULL,
          BRCH_STREET              CHAR(40)     NOT NULL,
          BRCH_CITY                CHAR(16)     NOT NULL,
          BRCH_STATE               CHAR(2)      NOT NULL,
          BRCH_ZIP                 CHAR(5)      NOT NULL,
          PRIMARY KEY (BRCH_ID)
```

BRANCH RESERVE Table

```
  CREATE TABLE BRANCHRESERVE
         (BRSV_ID                  CHAR(4)      NOT NULL,
          BRSV_BRID                CHAR(5)      NOT NULL,
          BRSV_TOTAL               FLOAT        NOT NULL,
          PRIMARY KEY (BRSV_ID),
          FOREIGN KEY BRSVBRCH (BRSV_BRID)
              REFERENCES BRANCH ON DELETE RESTRICT)
```

CASHIER Table

```
  CREATE TABLE CASHIER
         (CASH_ID                  CHAR(6)      NOT NULL,
          CASH_FNAME               CHAR(16)     NOT NULL,
          CASH_LNAME               CHAR(24)     NOT NULL,
          CASH_BRID                CHAR(4)      NOT NULL,
          PRIMARY KEY (CASH_ID)
          FOREIGN KEY CASHBRCH (CASH_BRID)
              REFERENCES BRANCH ON DELETE RESTRICT)
```

CASHIER DRAWER Table

```
  CREATE TABLE CASHIERDRAWER
         (CDRW_ID                  CHAR(4)      NOT NULL,
          CDRW_CAID                CHAR(6)      NOT NULL,
          CDRW_SIZE                CHAR(9),
          CDRW_SLOTS               INT,
          PRIMARY KEY (CDRW_ID),
          FOREIGN KEY CDRWCASH (CDRW_CAID)
              REFERENCES CASHIER ON DELETE RESTRICT)
```

COUNTRY Table

```
  CREATE TABLE COUNTRY
         (CNTY_NAME                CHAR(30)     NOT NULL,
          CNTY_ID                  CHAR(3)      NOT NULL,
          PRIMARY KEY (CNTY_NAME)
```

CURRENCY Table

```
  CREATE TABLE CURRENCY
         (CURR_ID                  CHAR(3)      NOT NULL,
          CURR_NAME                CHAR(16)     NOT NULL,
```

```
        CURR_CNTY               CHAR(30)    NOT NULL,
        CURR_XRATE_BUY          DEC(8,4)    NOT NULL,
        CURR_XRATE_SELL         DEC(8,4)    NOT NULL,
        CURR_FORG_INFO          CHAR(40),
        CURR_DESC               CHAR(80),
        PRIMARY KEY (CURR_ID),
        FOREIGN KEY CURRCNTY (CURR_CNTY)
            REFERENCES COUNTRY ON DELETE CASCADE)
```

CUSTOMER ORDER Table

```
CREATE TABLE CUSTOMERORDER
       (CORD_ID                 INT         NOT NULL,
        CORD_CREATED            CHAR(8)     NOT NULL,
        CORD_STATUS             CHAR(4)     NOT NULL,
        CORD_TOTAL              FLOAT       NOT NULL,
        CORD_CAID               CHAR(6),
        CORD_CUID               INT         NOT NULL,
        PRIMARY KEY (CORD_ID),
        FOREIGN KEY CORDCASH (CORD_CAID)
            REFERENCES CASHIER ON DELETE SET NULL,
        FOREIGN KEY CORDCUST (CORD_CUID)
            REFERENCES CUSTOMER ON DELETE RESTRICT)
```

CUSTOMER Table

```
CREATE TABLE CUSTOMER
       (CUST_ID                 INT         NOT NULL,
        CUST_ACID               CHAR(9),
        CUST_FNAME              CHAR(16)    NOT NULL,
        CUST_LNAME              CHAR(24)    NOT NULL,
        CUST_STREET             CHAR(40)    NOT NULL,
        CUST_CITY               CHAR(16)    NOT NULL,
        CUST_STATE              CHAR(2)     NOT NULL,
        CUST_ZIP                CHAR(5),
        CUST_HPHONE             CHAR(8)     NOT NULL,
        CUST_WPHONE             CHAR(8)     NOT NULL,
        PRIMARY KEY (CUST_ID),
        FOREIGN KEY CUSTACCT (CUST_ACID)
            REFERENCES ACCOUNT ON DELETE RESTRICT)
```

DENOMINATION Table

```
CREATE TABLE DENOMINATION
       (DNOM_ID                 CHAR(3)     NOT NULL,
        DNOM_TYPE               CHAR(8)     NOT NULL,
        DNOM_VALUE              INT         NOT NULL,
        DNOM_DESC               CHAR(60),
        PRIMARY KEY (DNOM_ID, DNOM_TYPE, DNOM_VALUE),
        FOREIGN KEY DNOMCURR (DNOM_ID)
            REFERENCES CURRENCY ON DELETE CASCADE)
```

ORDERITEM Table

```
CREATE TABLE ORDERITEM
       (ORDI_ORID               INT         NOT NULL,
        ORDI_ID                 CHAR(3)     NOT NULL,
        ORDI_TYPE               CHAR(8)     NOT NULL,
        ORDI_VALUE              INT         NOT NULL,
        ORDI_QTY                INT         NOT NULL,
        ORDI_FORGN_TOTAL        FLOAT       NOT NULL,
        ORDI_LOCAL_TOTAL        FLOAT       NOT NULL
        PRIMARY KEY (ORDI_ORID, ORDI_ID, ORDI_TYPE, ORDI_VALUE)
```

STOCKITEM Table

```
CREATE TABLE STOCKITEM
       (STKI_DRID               CHAR(4)     NOT NULL,
        STKI_ID                 CHAR(3)     NOT NULL,
```

```
STKI_TYPE                CHAR(8)      NOT NULL,
STKI_VALUE               INT          NOT NULL,
STKI_QTY                 INT          NOT NULL,
STKI_MIN                 INT          NOT NULL,
STKI_MAX                 INT          NOT NULL
PRIMARY KEY (STKI_DRID, STKI_ID, STKI_TYPE, STKI_VALUE),
FOREIGN KEY STKIDNOM (STKI_ID, STKI_TYPE, STKI_VALUE)
    REFERENCES DENOMINATION ON DELETE RESTRICT)
```

B.2 Relational System Build Programs

Subtopics

B.2.1 Database Creation REXX Command File

```
/* this is a REXX command file which will    */
/* create the CSOODB database                */

if Rxfuncquery('SQLEXEC') <> 0 then
  rcy = Rxfuncadd('SQLEXEC','SQLAR','SQLEXEC')
if Rxfuncquery('SQLDBS') <> 0 then
  rcy = Rxfuncadd('SQLDBS','SQLAR','SQLDBS')

say 'Starting DB2/2 Processing'
call SQLDBS 'START DATABASE MANAGER'
if (SQLCA.SQLCODE <> -1026) then
   rcy = ERROR();

prodName = 'VisualAge'
dbname = 'CSOODB'
codeOK = 'nil'

say 'Stopping any database currently in use'
call SQLDBS 'STOP USING DATABASE';
if (SQLCA.SQLCODE <> -1024) then
    rcy = ERROR();

say '     Dropping Database (' dbname ')'
call SQLDBS 'DROP DATABASE' dbname
if (SQLCA.SQLCODE = 0000) | (SQLCA.SQLCODE = -1013) then
      codeOK = 'yes'
else
      rcy = ERROR();

if (codeOK = 'yes') then
do;

  say '     Creating Database (' dbname ')'

  call SQLDBS 'CREATE DATABASE ' dbname

  if (SQLCA.SQLCODE <> 0000) then
    dbCreated = 'yes'
  else
    rcy = ERROR();

  say '     Opening DataBase (' dbname ')'

  call SQLDBS 'START USING DATABASE' dbname
  rcy = ERROR();

  say '     Closing Database (' dbname ')'
  call SQLDBS 'STOP USING DATABASE';
  rcy = ERROR();

  if (dbCreated = 'yes') then
  do;
  say ' '
  say 'A new database (' dbName ') was just created!!'
  say 'This database must be bound to the' prodName 'DLL!'
  end;
say 'Completed DB2/2 Processing'
end;
rcy = Rxfuncdrop('SQLEXEC')
rcy = Rxfuncdrop('SQLDBS')
exit;

ERROR:
  if (RESULT = 0 & SQLCA.SQLCODE = 0) then
  return 0;
  say 'RESULT        = ' RESULT;
  say 'SQLCA.SQLCODE = ' SQLCA.SQLCODE;
  say 'SQLMSG        = ' SQLMSG;

  call SQLDBS 'STOP USING DATABASE';
exit;
```

B.2.2 Table Creation REXX Command File

```
/* this is a REXX command file which will    */
/* drop then create a sample table           */

if Rxfuncquery('SQLEXEC') <> 0 then
  rcy = Rxfuncadd('SQLEXEC','SQLAR','SQLEXEC')
if Rxfuncquery('SQLDBS') <> 0 then
  rcy = Rxfuncadd('SQLDBS','SQLAR','SQLDBS')

say 'Start DB2/2 Processing'
call SQLDBS 'START DATABASE MANAGER'
if (SQLCA.SQLCODE <> -1026) then
   rcy = ERROR();

prodName  = 'VisualAge'
dbname     = 'CSOODB'
table   = substitute table name

call SQLStmts

say 'DB2/2 database disconnect'
call SQLDBS 'STOP USING DATABASE';
if (SQLCA.SQLCODE <> -1024) then
    rcy = ERROR();

say ' OPENING DATABASE (' dbname ')'
call SQLDBS 'START USING DATABASE' dbname
rcy = ERROR();

say '     Dropping Table (' itemTable ')'
call SQLEXEC 'PREPARE s1 FROM :dropTbl_stmt'
  rcy = ERROR();
call SQLEXEC 'EXECUTE s1'
if (SQLCA.SQLCODE <> 0000) & (SQLCA.SQLCODE <> -204) then
  rcy = ERROR();

say '     Creating Table (' itemTable ')'
call SQLEXEC 'PREPARE s1 FROM :crtTable_stmt'
  rcy = ERROR();
call SQLEXEC 'EXECUTE s1'
if (SQLCA.SQLCODE <> 0000) & (SQLCA.SQLCODE <> -601) then
  rcy = ERROR();

say ' CLOSING DATABASE (' dbname ')'
call SQLDBS 'STOP USING DATABASE';

rcy = Rxfuncdrop('SQLEXEC')
rcy = Rxfuncdrop('SQLDBS')
say 'Completed DB2/2 processing'
exit;

SQLStmts:

crtTable_stmt = Table DDL Statements
dropTbl_stmt  = ' DROP TABLE 'substitiute table name
return

ERROR:

if (RESULT = 0 & SQLCA.SQLCODE = 0) then
  return 0;
say 'RESULT                    = ' RESULT;
say 'SQLCA.SQLCODE  = ' SQLCA.SQLCODE;
say 'SQLMSG                   = ' SQLMSG;

call SQLDBS 'STOP USING DATABASE';
exit;
```

B.2.3 Table Load REXX Command File

```
/* this is a REXX command                    */
/* that will load i.e. populate a test table */

if Rxfuncquery('SQLEXEC') <> 0 then
  rcy = Rxfuncadd('SQLEXEC','SQLAR','SQLEXEC')
if Rxfuncquery('SQLDBS') <> 0 then
  rcy = Rxfuncadd('SQLDBS','SQLAR','SQLDBS')

table = substitute table name

say 'Start DB2/2 load processing'
call SQLDBS 'START DATABASE MANAGER'
if (SQLCA.SQLCODE <> -1026) then
   rcy = ERROR();

say '      Loading table ( 'table' )'
call SQLDBS
   'import to csoodb from a:\csdb\cnty.del of del insert
    into country messages cnty.txt'

if (SQLCA.SQLCODE <> 0000) & (SQLCA.SQLCODE <> 3107) then
   rcy = ERROR();

say 'Complete DB2/2 load processing'

ERROR:

if (RESULT = 0 & SQLCA.SQLCODE = 0) then
  return 0;
  say 'RESULT                   = ' RESULT;
  say 'SQLCA.SQLCODE = ' SQLCA.SQLCODE;
  say 'SQLMSG                  = ' SQLMSG;

  call SQLDBS 'STOP USING DATABASE';

exit;
```

:biblio id=refer head='Bibliography'.

COMMENTS Appendix D. ITSO Technical Bulletin Evaluation RED000
Object-Oriented Application Development
with VisualAge
in a Client/Server Environment

Publication No. GG24-4227-00

Your feedback is very important to help us maintain the quality of ITSO Bulletins. **Please print out this questionnaire, fill it out, and then return it using one of the following methods:**

- ☐ Mail it to the address on the back (postage paid in U.S. only)
- ☐ Give it to an IBM marketing representative for mailing
- ☐ Fax it to: Your International Access Code + 1 914 432 8246
- ☐ Send a note to REDBOOK@VNET.IBM.COM
- ☐ Copy this section to file and send it via VNET to: QUALITY @ WTSCPOK

Please rate on a scale of 1 to 5 the subjects below.
(1 = very good, 2 = good, 3 = average, 4 = poor, 5 = very poor)

Overall Satisfaction ____

Organization of the book	____	Grammar/punctuation/spelling	____
Accuracy of the information	____	Ease of reading and understanding	____
Relevance of the information	____	Ease of finding information	____
Completeness of the information	____	Level of technical detail	____
Value of illustrations	____	Print quality	____

Please answer the following questions:

a) If you are an employee of IBM or its subsidiaries:

Do you provide billable services for 20% or more of your time? Yes____ No____

Are you in a Services Organization? Yes____ No____

b) Are you working in the USA? Yes____ No____

c) Was the Bulletin published in time for your needs? Yes____ No____

d) Did this Bulletin meet your needs? Yes____ No____

If no, please explain:

What other topics would you like to see in this Bulletin?

What other Technical Bulletins would you like to see published?

Comments/Suggestions: **(THANK YOU FOR YOUR FEEDBACK!)**

Address your comments to:

IBM International Technical Support Organization
Department 471, Building 070B
5600 COTTLE ROAD
SAN JOSE CA
USA 95193-0001

Name . . . . ____
Company or Organization ____
Address . . . . ____

____
____
Phone No. . . . . ____